Spring 5-22-2015

FIREFOX ADD-ON FOR METAMORPHIC JAVASCRIPT MALWARE DETECTION

Sravan Kumar Reddy Javaji
San Jose State University

FIREFOX ADD-ON FOR METAMORPHIC JAVASCRIPT MALWARE DETECTION

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Sravan Kumar Reddy Javaji

May 2015

The Designated Thesis Committee Approves the Thesis Titled

FIREFOX ADD-ON FOR METAMORPHIC JAVASCRIPT MALWARE DETECTION

by

Sravan Kumar Reddy Javaji

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Mr. Fabio Di Troia	University of Sannio

ABSTRACT

Firefox Add-on for Metamorphic JavaScript Malware Detection

by Sravan Kumar Reddy Javaji

With the increasing use of the Internet, malicious software has more frequently been designed to take control of users computers for illicit purposes. Cybercriminals are putting a lot of efforts to make malware difficult to detect. In this study, we demonstrate how the metamorphic JavaScript malware can effect a victim's machine using a malicious or compromised Firefox add-on. Following the same methodology, we develop another add-on with malware static detection technique to detect metamorphic JavaScript malware.

ACKNOWLEDGMENTS

I am very thankful to my advisor Dr. Thomas Austin for his continuous guidance and support throughout this project and believing me. Also, I would like to thank the committee members Dr. Chris Pollett and Mr. Fabio Di Troia for monitoring the progress of the project and their valuable time.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The arrival of the Internet has completely revolutionized our personal and professional lives. With the rapid growth of the Internet, all the market sectors, social networking services, advertising and non-commerical sectors are using this technology in their workflow. As we become more dependent on the online environment, we can see massive growth of opportunities for IT criminals to take advantage of user systems.

Internet users often share sensitive information like bank account details or other personal information, over the network. As personal computers and mobile phones became an important part in most people‘s lives, these computers became a hub of user‘s personal information. In this world of ubiquitous computers and persistent threats from hackers, protecting your computer is a must. Several websites are hacked to be used as distributors of malware, to infect the visitors unknowingly with viruses and malware. A single visit to a such a hacked web page is sufficient for an intruder to get control of a user‘s machine.

In late 2013, one of the bank‘s internal computers that are used by employees to process and record daily transactions, had been infected with malware [7]. The malware continuously monitored the bank‘s activities for several months, sending back images and video feeds to cybercriminals about the bank‘s daily routines. Then the cybercriminals impersonated bank officers, turned on several cash machines and also transferred millions of dollars from banks into dummy accounts of other countries.

Consider the fact that more than 6,600 benign websites are getting hacked every single day [6]. These legitimate websites are turned into distributors of malware by malicious hackers. Malicious code can be injected into legitimate Javascript of a benign web page. When a user visits such a compromised website, this malicious JavaScript will be executed in the victim‘s web browser. Execution of such malicious JavaScript can infect the victim‘s personal computer. Most of the times, malicious JavaScript redirects the victim‘s web browser to load more malicious code from a remote server. This can be achieved through several means, such as adding an HTML iframe element to a page. Always cybercriminals try to obfuscate the malicious content from detection. HTML provides very few ways to obfuscate the code such as adding an HTML iframe element to a page but the huge number of methods in JavaScript makes it easy to heavily obfuscate the malicious code into Javascript.

1.1 Problem

Malware is malicious software, specifically designed to gain access to the data or to damage the resources without the knowledge of victim [4]. Researchers developed various techniques for malware detection like signature based detection and heuristic analysis. To overcome the malware detection techniques, malware writers came up with different types of malwares among which metamorphic malware is an advanced version. Metamorphic malware is capable of changing its internal structure without altering its functionality from infection to infection. Due to the metamorphic nature, such malware is very difficult to detect. With a huge set of functions, JavaScript makes it possible for virus writers to develop malicious JavaScript code with metamorphic feature. Transcriptase is such a case, which is

```
//sample.js
<script>
    var x = 5;
    var y = 6;
    var z = x + y;
</script>
```

```
//sample_morphed.js - obfuscated version of sample.js
<script>
    var y_renamed = 6;
    var x_renamed = 5;
    var z_renamed = x_renamed + y_renamed;
</script>
```

Figure 1: JavaScript Sample and its obfuscated version.

designed to infect other JavaScript files in the same folder [5].

1.2 Proposed Solution

In spite of the fact that malware can change their internal structure the priority order of the important commands cannot be changed. Case in point, consider a sample JavaScript code below,

From the code in Figure 1, it is clear that even though the variable names are changed and the order of declaration of x and y (or $x\_renamed$ and $y\_renamed$) are changed, the arithmetic operation always follows the declaration of those two variables else the code may give syntactical errors or the wrong result.

So, we can make use of the opcodes statistical information to detect the malware.

1.3 A Browser Plugin for Detecting Malware

Generally JavaScript malware will be injected into benign web pages. When a user visits this infected website, the malicious JavaScript code will be executed in the browser. To prevent this, we can develop an add-on which will monitor the JavaScript content of every web page and the browser can disable the JavaScript execution before the page gets loaded. The add-on will analyze the JavaScript in the background and will enable the JavaScript load, if the web page is found to be benign or else warn the user about malicious content without loading the JavaScript. This procedure can secure the victim‘s computer from malware infection. For this research, we can use the Transcriptase metamorphic JavaScript malware.

The remaining of the paper is organized as follows. In Chapter 2, we provide background information on metamorphic malware and with an emphasis on Transcriptase metamorphic JavaScript malware that forms the basis for the research. Also in this chapter, we cover the Rhino Javascript engine. Chapter 3 outlines the details of Firefox add-on development. Then in Chapter 4, we discuss two different static metamorphic detection techniques that we apply to detect the metamorphic JavaScript malware. In Chapter 5, we present the accuracy and performance details of Firefox add-on using opcode similarity detection technique. Our experimental results for the original metamorphic JavaScript appear. Chapter 6 contains the conclusion and consideration for future work.

CHAPTER 2

Background

Malware is a software program intended to do pernicious activities on a client‘s computer with the proposition of removing data and misusing assets without his assent. Viruses and worms are the best known types of malware on account of the way in which they spread, instead of their behavior. Malware is now and then utilized widely against government or corporate websites to gather protected data or to disturb their operations by large. Also, malware is regularly utilized against people to steal data, for example, personal identification numbers or bank or credit card details and passwords. As per a survey on data breaches led by Verizon in 2014 [9], Citadel is the preferred banking malware among attackers for stealing individual information. And for stealing money from bank accounts, Zeus is the favorite banking malware.

Figure 1 shows how the malware is swiftly growing in volume day-by-day. In Figure 1, the x-axis specifies the year and the y-axis indicates the number of malware samples generated in the specified year on the x-axis.

Virus writers are aware that signature-based detection with heuristic analysis can be the basis of modern malware detection techniques. So, virus developers have created numerous procedures and techniques to evade signature-based detection. In January 2015, AV-TEST‘s CEO said, "Many of the new malware samples are just variants of existing viruses. They have been modified so that they are no longer detected and thus, AV signature updates are required" [10]. Some of the noteworthy techniques used by virus writers to evade signature detection are encryption,

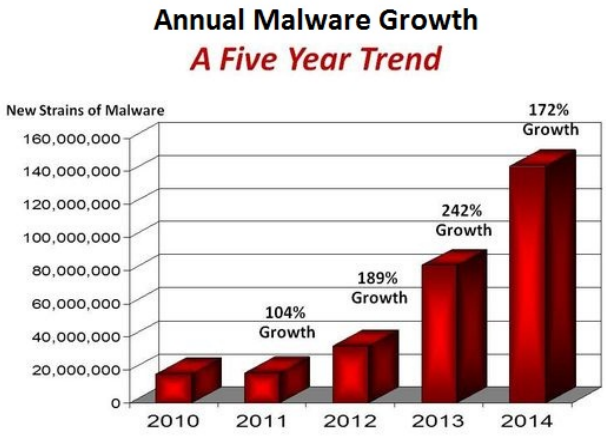

Figure 2: Annual Malware Growth [10]

polymorphism and metamorphism.

2.1 Encrypted Malware

The Cascade virus, which initially appeared in late 1986, was the first malware that used encryption to scramble its contents [11]. The Cascade virus is comprised of two parts. The first part is a decryptor and the second part is encrypted malware code. The reason for encryption is to conceal the malware signature, so as to evade signature identification. Later, this technique was adopted by almost every encrypted malware. For the most part, virus writers use extremely simple and weak encryption methods, for example, a repeated XOR with a fixed bit pattern. Cascade malware also used XOR operation as encryption routine because of its symmetrical and reversible feature. In the event that the encryption key of malware was changed after every infection, the encrypted body signature also gets changed. If in case the same decryptor was used, signature detection can make use of the decryptor code‘s signature to detect the malware.

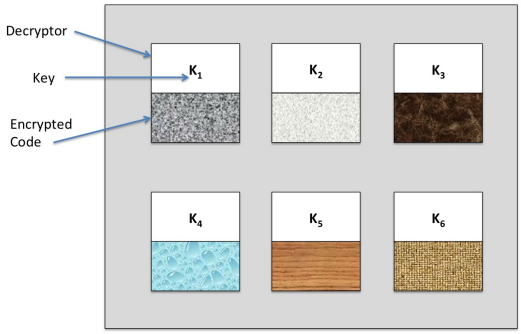

Figure 3: An Encrypting malware spreads without changing decryptor but the key within decryptor varies from infection to infection. As the key value changes, the encrypted virus body also changes [14].

2.2 Polymorphic Malware

Similar to an encrypted malware, polymorphic malware incorporates an encrypted virus code and a decryptor. Additionally in a polymorphic virus, the decryptor is morphed. During polymorphic malware propogation, not only is the virus code encrypted, but the decryptor also varies from infection to infection. As there is no fixed signature or no fixed decryptor to scan for, no two infections look alike to be exploited by the antivirus program for detection purpose [14]. Polymorphic virus uses code obfuscation techniques, for example, including junk codes or substitution of instructions, to mutate its decryptor [18]. Several techniques are utilized to decrypt the polymorphic virus, such as, cryptanalysis (also called x-ray), emulation and dedicated decryption routines [21].

The first polymorphic malware, 1260, was developed by Mark Washburn in 1990 [15]. And the first widespread polymorphic infection was caused by Tequila and Maltese Amoeba virus, in 1991 [14].

```
MOV A,R1        MOV A,R1        MOV A,R1        MOV A,R1        MOV A,R1
ADD B,R1        NOP             ADD #0,R1       OR R1,R1        TST R1
ADD C,R1        ADD B,R1        ADD B,R1        ADD B,R1        ADD C,R1
SUB #4,R1       NOP             OR R1,R1        MOV R1,R5       MOV R1,R5
MOV R1,X        ADD C,R1        ADD C,R1        ADD C,R1        ADD B,R1
                NOP             SHL #0,R1       SHL R1,0        CMP R2,R5
                SUB #4,R1       SUB #4,R1       SUB #4,R1       SUB #4,R1
                NOP             JMP .+1         ADD R5,R5       JMP .+1
                MOV R1,X        MOV R1,X        MOV R1,X        MOV R1,X
                                                MOV R5,Y        MOV R5,Y
(a)             (b)              (c)             (d)             (e)
```

Figure 4: Examples of a polymorphic virus utilizing code obfuscation techniques. All of the Figure 4 snippets perform the same operation, i.e., $X = (A + B + C - 4)$. For instance, the program snippet of Figure 4 (c) is functionally the same as Figure 4 (a) in light of the fact that instructions like adding 0 to a register, ORing R1 with itself, shifting R1 left 0 bits, and jumping to the next instruction all do nothing [19].

2.3 Metamorphic Malware

Virus writers took the next step and developed an advanced variant of polymorphic malware, known as metamorphic malware. Generally, before infection, polymorphic malware encrypt the virus code and morph the decryptor, while metamorphic malware morph the whole virus code. According to Igor Muttik, "Metamorphics are bodypolymorphics", since polymorphism is applied to the entire virus body [22]. Metamorphic viruses utilizes several code morphing techniques that constitute instruction reordering, data reordering, subroutine inlining, subroutine outlining, register renaming, instruction substitution and dead code insertion [23]. Figure 5 illustrates the metamorphic malware with different signatures.

2.3.1 Register renaming

In December 1998, a metamorphic malware named Win95/Regswap was developed by Vecna [22]. Regswap used register renaming technique to morph the

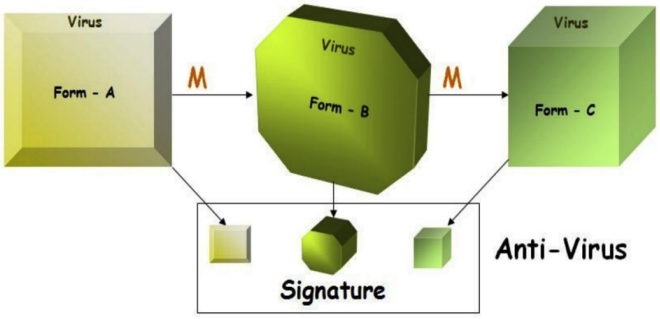

Figure 5: Metamorphic malware with different signatures [20].

```
a)
5A                pop edx
BF04000000        mov edi,0004h
8BF5              mov esi,ebp
B80C000000        mov eax,000Ch
81C288000000      add edx,0088h
8B1A              mov ebx,[edx]
899C8618110000    mov [esi+eax*4+00001118],ebx

b)
58                pop eax
BB04000000        mov ebx,0004h
8BD5              mov edx,ebp
BF0C000000        mov edi,000Ch
81C088000000      add eax,0088h
8B30              mov esi,[eax]
89B4BA18110000    mov [edx+edi*4+00001118],esi
```

Figure 6: code snippet extracted from two different versions of RegSwap [22].

virus code. In this technique, instructions gets modified to use different registers. As just the register operands gets altered that too in some part of the code instead of whole code, so the complexity of final modified code wouldn't be high. Figure 6 depicts how the register renaming technique transforms the code.

The bold areas in figure 6 illustrates the similarities of the two different code versions. Thus, a wildcard string, such as $5?B?$, could be useful to detect the

```
C7060F000055     MOV [esi], 5500000Fh
C746048BEC5151   MOV [esi+0004], 5151EC8Bh
BF0F00055        MOV edi, 5500000Fh
893E             MOV [esi], edi
5F               POP edi                ; garbage
52               PUSH edx               ; garbage
B640             MOV dh, 40             ; garbage
BA8BEC5151       MOV edx, 5151EC8Bh
53               PUSH ebx               ; garbage
8BDA             MOV ebx, edx
895E04           MOV [esi+0004], ebx
```

Figure 7: Example of code metamorphosis of Evol virus with Dead code insertion [23].

malware code [22].

2.3.2 Dead code insertion

Dead code can be a single instruction or a block of instructions, for example, adding NOPs [23], adding 0 to a register, moving between same registers, ORing register with itself, shifting register left 0 bits, jumping to next instruction, incrementing a register immediately followed by decrementing the same register by same value. Inserting dead code or do-nothing instructions is the easiest approach to morph the virus code without modifying its functionality [23]. The Win32/Evol virus, which was found around July 2000 [24], used dead code insertion to obfuscate the signature of a code as illustrated in Figure 7.

2.3.3 Subroutine permutation

Code may contain several subroutines (or) functions and changing the order of this subroutines may not impact the execution of code. Subroutine permutation approach makes use of this advantage i.e., altering the order of subroutines, to

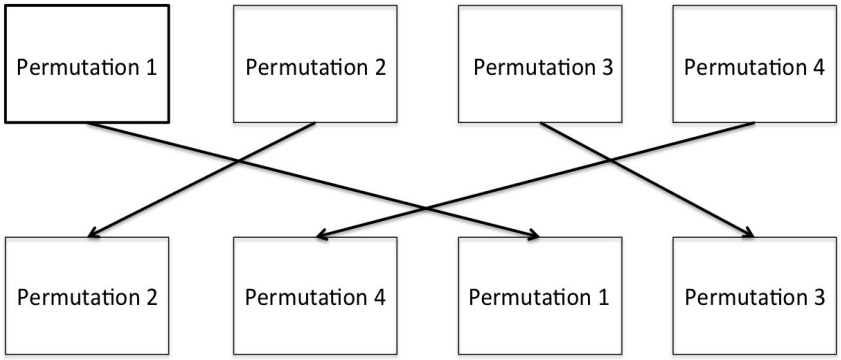

Figure 8: Example of subroutines permutation [23].

change its internal structure without modifying the functionality of code. A code with n different subroutines can generate $n!$ different permutations of subroutines, thus large number of versions of the same code can be generated [4].

2.3.4 Equivalent code substitution

Different variants of code can be generated by replacing instruction or block of instructions with an equivalent code. In assembly language there are numerous semantically equivalent instructions, for instance, 'INC ecx' is same as 'ADD ecx, 1', 'XOR R1, R1' is same as 'MOV R1, 0' [25].

2.3.5 Transposition

Morphed copies of virus code can also be created by changing the order of instructions in the code provided that there is no dependency among instructions, so this approach is also known as instruction permutation [23]. For instance, the code in figure 9 can be transformed to figure 10, as the declaration order of variables doesn't affect the arithmetic calculation [4].

```
1  int a=5;
2  int b=2;
3  int c=a+b;
```

Figure 9: Sample Java code [4].

```
1  int b=2;
2  int a=5;
3  int c=a+b;
```

Figure 10: Sample Java code after swapping instructions [4].

2.3.6 Changing control flow

The next code obfuscation method involves insertion of a conditional or unconditional branch instruction after a block of instructions. Further, instruction blocks referenced by this branching instructions can be permuted to change the control flow [23]. Zperm malware used this approach to change the internal structure of a code [25]. Figure 11 is an example of changing control flow.

2.3.7 Subroutine inlining and outlining

In subroutine inlining procedure, subroutine/function call replaces its code [23]. Figure 12 illustrates the concept of Subroutine inlining.

On the other hand, code outlining changes a block of instructions into subroutine (or function) and a subroutine call will be included for the newly created subroutine. Figure 13 illustrates how the code outlining approach works.

2.4 Transcriptase

Transcriptase is a metamorphic virus implemented in JavaScript. Whenever Transcriptase is executed, a morphed version of the malware virus gets prepended

```
;Original Program
instruction 1  ; entry point
instruction 2
instruction 3
instruction 4
instruction 5
```

```
;Modified Program
instruction 2
jump 3
instruction 4
jump 5
instruction 1  ; entry point
jump 2
instruction 3
jump 4
instruction 5
```

Figure 11: Example of changing control flow [25].

```
   /* some instrictions */        /* some instrictions */
   call S1                       mov eax, ebx
   call S2                       add eax, 12h
   /* some instrictions */        push eax
   S1:                           mul ecx
    mov eax, ebx                 mov edx, eax
    add eax, 12h                 /* some instrictions */
    push eax
    ret
   S2:
    mul ecx
    mov edx, eax
    ret
(a) before transformation     (b) after transformation
```

Figure 12: Subroutine inlining example [25].

to all the JavaScript files in the folder [4]. The infected JavaScript file will become the variant of Transcriptase. By this way Transcriptase propagates and infects the benign JavaScript files.

```
   /* some instrictions */          /* some instrictions */
   mov  eax, ebx                   mov  eax, ebx
   add  eax, 12h                   call S1
   push eax                        mov  edx, eax
   mul  ecx                        /* some instrictions */
   mov  edx, eax                   S1:
   /* some instrictions */          push eax
                                    add  eax, 12h
                                    mul  ecx
                                    ret
(a) before transformation       (b) after transformation
```

Figure 13: Subroutine outlining example [25].

The metamorphic engine attached to Transcriptase is a self-hosted compiler, which contains its own meta-language source-code. Transcriptase obtains information of its code from meta-language and changes its internal structure.

The format of every line of the meta-code looks like [26]:

```
(Identifier|Restrictions)instruction
```

For instance, below are sample meta-instructions:

```
(200|)var b=0
(300|200)c+1(b)
```

An Identifier and Restrictions are used by the Permutation function to do code obfuscation. The identifier is unique for every instruction in the entire code and restrictions specify the instructions on which the corresponding instruction is depending. The "instruction" contains the details used to create an actual code.

The compiler creates the new malicious JavaScript code with three steps:

1. Permutation and Variable/Function-Name randomization

```
(100|)var a=5
(200|)var b=-1
(300|)var x=8
(400|200)c+1(b) // instruction c+1(b) means increment b by 1: i.e. b++
(500|400,100)c+n(b,a) // instruction c+n(b,a) means increment b by a:
    i.e. b+=a
(600|500)xWScript.Echo(x)
```

Figure 14: Meta-language instructions example [4].

2. Code Creation

3. Variable/Function insertion

2.4.1 Permutator

In this phase, the compiler parses through every meta-language instruction, scope by scope (global scope for global instructions and sub-scope for if/for/while/functions) and retrieves the identifiers and restrictions details for each meta-instruction. Later these identifier and restriction details are used by the compiler to perform the permutation of code.

If the restriction details are empty for all the meta-language instructions, then it specifies that all the instructions in the code do not have any dependency instructions. So, the entire code can be permuted in all the possible combinations. For instance, if there are n lines of code, then the permutator can create $n!$ variations of the original code.

For instance, consider the code specified in Figure 14 [4]. It contains restriction details for some of the meta-instructions, which means that those instructions have dependencies on other instructions. So all the combinations of the code cannot produce the correct behaviour.

```
(200|)var b=-1
(400|200)c+1(b)
(100|)var a=5
(500|400,100)c+n(b,a) // instruction c+n(b,a) means increment b by a:
    i.e. b+=a
(300|)var x=8
(600|500)xWScript.Echo(x)
```

Figure 15: Possible output of the permutator after parsing the code in Figure 14 [4].

From the code in Figure 14, instruction 400 depends on instruction 200, because the variable "b" has to be defined before it can be incremented; instruction 500 depends on both the instructions 400 and 100; and instruction 600 depends on instruction 300. Figure 15 contains the code, which could be one of the possible output generated by the permutator [4]:

As the growth-rate of the permutation function is very fast, even for the large number of instruction this technique works effectively [26].

2.4.2 Variable/Function-Name randomization

In this step, the keywords like "var", "while", "for", and "def" are searched initially in the code by the compiler and the details of existing hard-coded names in those instructions are retrieved. In other words, the details of all the variable names and function names are gathered by the compiler. These names are replaced with random names by the compiler and also all the valid occurrences of these hard-coded names in the current scope are replaced.

For instance, consider the code in Figure 16. First, the compiler searches for the "var", "function", and "def" keywords and the hard-coded names - num, multiply, inputparam, and twiceval are retrieved. Then it replaces these hard-coded

```
var num=20;
function multiply(inputparam)
{
   return 2 * inputparam;
}
def twiceval=multiply(num)
```

Figure 16: Before Variable/Function-Name randomization [26].

```
var ljkjuytbenst=20;
function trqwsdexcv(awsrsfagfqwxczv)
{
   return 2 * awsrsfagfqwxczv;
}
def bxswdqtyzyqtc=trqwsdexcv(ljkjuytbenst)
```

Figure 17: After randomly changing the Variables/Function-Names of the code in Figure 16 [26].

names with some random names like "ljkjuytbenst" or "awsrsfagfqwxczv". The compiler also replaces all occurrences of the hard-coded names, for example, there are two instances of "inputparam" present in the function, both these names are replaced with the random name. One of the possible changes to the code during this phase is shown in Figure 17.

2.4.3 Meta-Language Symbols

After rearrangement of the instructions and hard-coded names replacement, the compiler generates a valid JavaScript code by parsing through every meta-instruction. These meta-instructions contain meta-level symbols and each of these symbols has specific meaning like number, element, object etc. While parsing meta-instructions, compiler processes these meta-level symbols.

For example, consider the code in Figure 18. Here meta-symbol #n...n#

```
var number=#n1n#
var str=#"Hello VXers"#
var exp=#x1true1x#
x#O1WScript#.Echo(°+str+°)1O#
```

Figure 18: Meta-Language Symbols Example [26].

```
x#O1WScript#.Echo(°+str+°)1O#

could become

function SomeFunction(SomeArg){WScript.Echo(SomeArg);}
SomeFunction(str)
```

Figure 19: Meta-Language Symbols processing Example [26].

specifies any value present in between #n's (i.e., in place of ...) specifies Number, meta-symbol #"..."# specifies any value present in the place of dots (or ...) specifies string, any value in #xN...Nx# specifies elements, the values between #01... #. ...10# specifies Objects and the symbol "°+ ... +°" specifies that the variable inside must be given as an argument for a function, if the instruction is derived into a function [26].

Figure 19 illustrates how the meta-symbols with objects are processed.

2.4.4 Code Derivation

After processing all the meta-language symbols, the compiler generates JavaScript code by parsing the meta-language instructions. During this phase, compiler deals with some more meta-instructions that have specific properties as mentioned below [26]:

while(initial$var1!var2?operator@action)NNN

1. "initial" specifies the code that is to be executed before the while loop like variable declaration
2. "var1" and "var2" along with the "operator" specifies the while loop condition.
3. "action" specifies the end of the loop instruction like counter increment.
4. "NNN" specifies total number of lines in the loop

cO(1||n||s)

This specifies general way of representing number/string arithmetic instruction. "O" specifies operator like +, -, *, /. Below are some of the meta-instructions that follow this format,

1. c+1(var1): increment var1 by 1
2. c+n(var1,var2): increment var1 by the number var2
3. c+s(var1,var2): concatenate var1 with the string var2
4. c-1(var1): decrement var1 by 1
5. c-n(var1,var2): decrement var1 by the number var2
6. c*1(var1): multiply var1 by 1
7. c*n(var1,var2): multiply var1 by the number var2
8. c/1(var1): divide var1 by 1
9. c/n(var1,var2): divide var1 by the number var2

2.4.5 Variable/Function insertion

Several variables and functions are defined during the compilation phase because of meta-instructions and obfuscation. These variables are saved in special

arrays instead of being stored in the code. At the end of code derivation phase, they are placed into the code .

Functions can be included between instructions in the global scope. Variables can be included between instructions in the current scope, before they are used for the first time [4]. This phase takes lot of time to complete, as the whole code is checked for multiple times to find suitable positions for the variable/function insertions [26].

2.5 Rhino

During 1997, Netscape began working on developing a variant of Netscape Navigator written in Java [27]. In order to implement the navigator in Java, they built a JavaScript engine entirely in Java, named Rhino. Rhino is open source software and is currently maintained by Mozilla.

Most of the time, JavaScript is utilized as a part of HTML for making interactive webpages. Anyhow, Rhino is not used to create or manipulate webpages; it is an implementation of the core JavaScript. Rhino has the below aspects [27]:

1. Supports JavaScript 1.7 features
2. Allows direct scripting of Java
3. The Rhino Shell can execute the JavaScript code interactively or in batch mode
4. The JavaScript Compiler can compile the JavaScript code into Java classes
5. The JavaScript Debugger for debugging scripts in Rhino

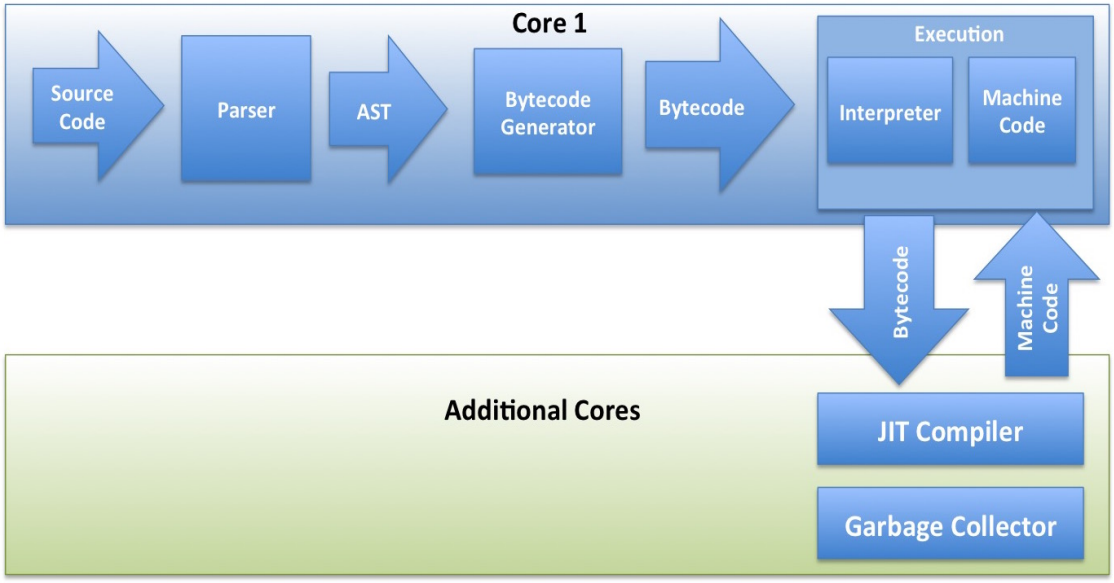

Figure 20: Block Diagram of Rhino Engine [29].

In this research, we use Rhino to translate the JavaScript code into Java classes. The engine supports both compile mode and interpretive mode. During compile mode, the engine first translates each JavaScript file to separate Java class files. These .class files may be executed as Java programs using Rhino runtime support routines. During interpretive mode, JavaScript is compiled and is stored as internal representation of the compiled form instead of byte codes. During runtime this compiled form is evaluated using rhino functions.

2.5.1 Architecture

The four basic blocks in the Rhino JavaScript engine are - the parser, the byte-code generator, the interpreter and the JIT [4]. Figure 20 depicts the block diagram of the Rhino Engine [29].

Parser: The input for this module is JavaScript code and the output generated is the Abstract Syntax Tree (AST). The AST is a tree representation of

```
function gcd(a, b)
{
   while (b != 0)
    {
      if a > b
         a -= b
        else
         b -= a
    }
    return a
}
```

Figure 21: Euclid's GCD algorithm.

the abstract syntactic structure of a program. For instance, Figure 21 represents the AST for the GCD code in Figure 22 [30].

The non-leaf nodes in the AST specify the operations to be performed, for instance, equal, comparison, arithmetic operation and so on. The leaf nodes in AST specify operands in the source code, for instance, a and b variables [4].

Byte-Code Generator: The AST output generated from the parser acts as input to the Byte-Code Generator which then converts the tree into byte code. During the conversion process, the Byte-Code generator picks each code block in the tree and translates it into bytecode. For instance, the byte code for the instruction $c = a - b$ is shown in Figure 23 [4].

The bytecode in Figure can be interpreted as -

1. Load the values that are stored at offset 1 and offset 2 into registers
2. Subtract these loaded values
3. Store the result at offset 3.

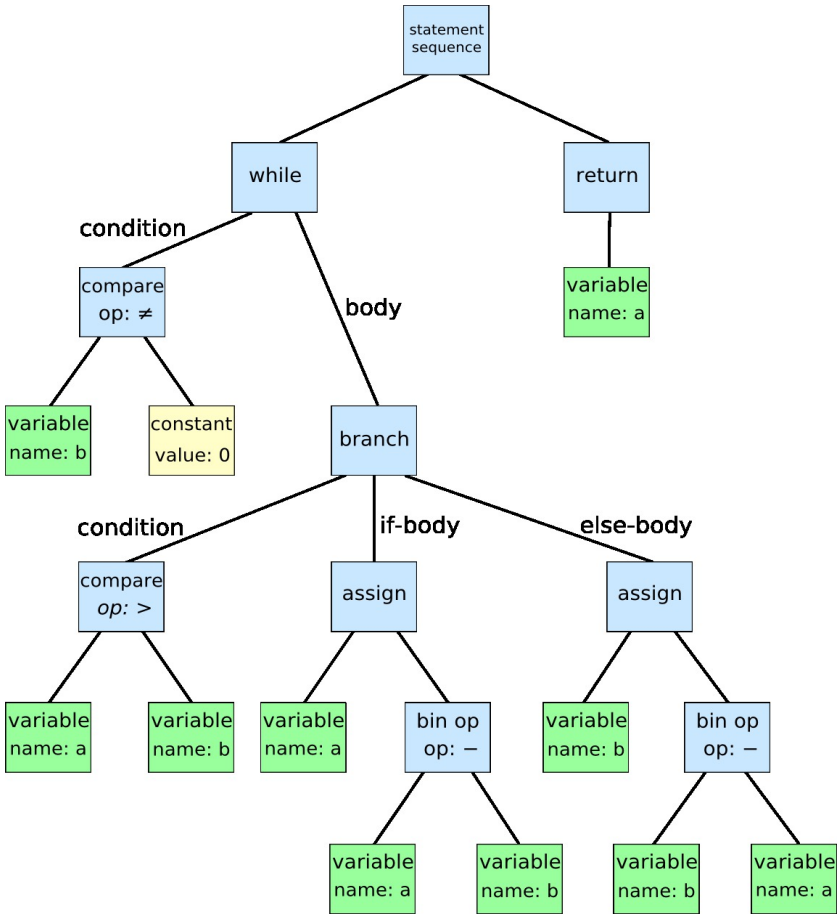

Figure 22: Abstract Syntax Tree for the GCD code in Figure 21 [30].

```
1 iload_1
2 iload_2
3 isub 3
4 istore_3
```

Figure 23: Sample byte code for the instruction $c = a - b$ [4].

Interpreter: The input for the interpreter is the byte code output of the Byte-Code Generator. The byte code is then converted into machine level code using the Just-in-time (JIT) compiler. During runtime, this generated machine code gets executed. When the byte code in Figure 23 is given as input to the Interpreter, it generates the machine code as shown in Figure 24.

```
MOV EAX 0xFF20
MOV EBX 0xFF24
SUB EAX EBX
MOV ECX EAX
```

Figure 24: Sample Machine Code generated for the byte code in Figure 23 [4].

2.5.2 Modification

For this research, we need the opcodes of JavaScript code, so we used the version of Rhino that was modified by the author of [4].

Below changes were made in the modified version [4]:

1. The page load time is directly proportional to compiling time of JavaScript, so Rhino was optimized to convert only small part of JavaScript files. Because of this optimization, the original statistics of the JavaScript code that will be used as part of analysis will get affected. In order to solve this problem the engine was modified to compile JavaScript of any length.

2. Different optimization techniques were used to optimize class files for speed execution. As these optimization techniques also affect the statistics of JavaScript code, they were disabled.

3. Generally opcodes are generated by decompiling the class files. This method consumes a lot of time as this process require the generation of class file and again decompilation of these class files. As the class files generated are of no use, the opcodes are extracted from the code during compilation itself. By following this approach, the time used for decompilation of class files is saved.

4. As the opcodes are extracted before the class file is created, this modification also solved the problem associated with class files optimization.

Figure 25: JavaScript compilation with Rhino.

5. The opcodes are tapped and redirected to the standard output. Thus all the opcodes are printed on the screen. This output can also be saved in a file using the unix redirect operator (>).

   The below command is used to run the Rhino engine:

```
java -cp <path_to_rhino_js.jar> org.mozilla.javascript.tools.jsc.Main
   <JavaScript_File_Path>
```

Figures 25 and 26 are the screenshots of compiling JavaScript code with the original version and the modified version of Rhino respectively.

Figure 26: JavaScript compilation with modified version of Rhino.

CHAPTER 3

Firefox Add-on Development

An add-on is a piece of software that augments another application. Based on the browser, different terms are used to refer to this software, like add-on, plug-in, or extension. An add-on cannot be executed as stand-alone software. Add-ons are used for different purposes like blocking advertisements and popups, downloading videos, and also to integrate several social network sites.

Below are some of the applications of add-ons

1. An add-on can change the browser interface, which includes changing themes, the look and feel of buttons, the menu bar, and tabs.

2. They are also capable of adding new features to the browser, like providing easy usage of various softwares in the form of toolbars.

3. Add-ons can also modify the behavior of browsers, like customizing the search option or page redirection.

4. Add-ons used as plug-ins let the browser support internet content. These include Flash, Silverlight, and music players like QuickTime, real player, online games, and many more.

5. On many browsers, online privacy is protected using add-ons. There are many types of add-ons that help to control and secure browsing and avoid attacks like preventing the user movements tracking on the browser.

Browser Extensions are supported by Microsoft Internet Explorer starting with version 5 released in 1999. From 2004, Mozilla Firefox supports add-ons. The Opera browser supports extensions starting with the desktop version 10 which was released in 2009. Google Chrome and Safari added support for extensions in 2010. Each browser has different variations in the browser extension syntax and they are compatible to their browser alone i.e., an extension built for one browser doesn‘t work on another. For using search engine tools irrespective of browsers, a project named ‘Mycroft’ [13] has been proposed, which is a database of over 20,000 search engine add-ons supported by multiple browsers.

3.1 Firefox vs Chrome

In new era, out of all the browser extensions, Firefox and Chrome are most well known because of their popularity, security, and appearance. Below are some of the notable comparisons between them regarding extensions:

1. Firefox has an outstanding extension base, i.e., https://addons.mozilla.org/en-US/firefox/, that offers more capable add-ons compared to all other browsers.

2. Firefox add-ons are very powerful and can perform anything that a Firefox process allows. Security features can be integrated into a Firefox add-on in a much more effective manner than Chrome extensions. So, it is possible to develop more advanced add-ons in Firefox, which would not be achievable on different browsers. Unlike Firefox, Chrome does not trust extensions completely and they provide very constrained APIs. For instance, without the user‘s approval, extensions in Chrome can‘t access the resource present outside of Chrome‘s sandbox, but a Firefox add-on can access the resource in the

filesystem without the user‘s permission.

For example, even though there are many Chrome extensions like ScriptSafe, NoScript Lite, which are similar to Firefox’s "NoScript", till now no chrome extension is able to provide all the features of Firefox’s NoScript because of the Chrome’s constrained extension APIs.

3. By providing constrained extension APIs, Chrome presents a permission system and restricts its extensions a bit more for security. Whereas in Firefox, as the add-ons has more privileges, there are chances of infecting a victim‘s machine. At worst, we may have to re-install the operating system to undo the effect created by a malicious add-on. To avoid these potential issues and to ensure that the add-ons are safe to install, they are manually reviewed before they publicly appear in the Mozilla add-ons gallery.

To detect JavaScript malware, we have to generate opcodes for the JavaScript code in the webpage and to save these opcodes we may need access to the user‘s filesystem and also we need to execute some other external scripts to validate the JavaScript code based on the saved opcodes. These tasks are only possible with the powerful APIs provided by Firefox, so we decided to implement a Firefox add-on to detect metamorphic malware.

3.2 SDK vs XUL

There are two main ways to build Firefox extensions. The traditional way is using XPCOM (Cross Platform Object Model) and XUL (XML User Interface Language). Much of Mozilla‘s documentation is focused on XUL add-ons, because this has been around for many years. More details about XUL add-on development

can be found at [32]. The add-on SDK is the newer kind and was built under the Jetpack Project. Jetpack‘s main agenda is to make it easy to build Firefox add-ons by using HTML, CSS, and JavaScript [3].

It is advisable to use the Add-on SDK because of the advantages it provides compared to XUL [33]:

1. Simplicity: High-level JavaScript APIs provided by the SDK like basic user interface components and their functionality simplify all the common tasks in add-on development.

2. Compatibility : Electrolysis [31], also called e10s, is the project under which Firefox is being developed with a new multiple process architecture. The API‘s provided by this SDK are designed to be forward-compatible with this new architecture.

3. Security: It is not easy to build insecure add-ons using the SDK. Even the insecure Add-on that was compromised can do much less damage to the victim‘s machine.

4. No restarts required: To install extensions developed using the SDK, we do not need to restart the browser.

5. Mobile Support: Add-ons can be developed for Firefox mobile using the experimental support provided by SDK 1.5

However, XUL provides a huge number of options for the UI when compared to the SDK, and that‘s the reason XUL is used for developing add-ons that require a rich user interface.

In this research, all the add-ons are built using the SDK as it provides simple APIs for developing most of the common tasks.

3.3 Chrome Authority Usage

Chrome Authority has nothing to do with Google Chrome. The Mozilla Developer Network(MDN) defines "**chrome**" as any visible parts of a browser other than the web pages, For instance, tabs, menu bar, and toolbars.

From the beginning of developing the SDK, it was assumed that developers may need to access the underlying browser (or) XPCOM services. So, the add-on SDK was developed to provide "chrome privileges" to the most powerful low level APIs. The "chrome privileges" grants low-level APIs to access the Components object that gives unrestricted access to the user system.

With chrome privileges, an add-on can perform any function the browser is capable of. These privileges can be obtained by the add-on using the "chrome" module as shown below [1]:

```
var {Cc, Ci} = require("chrome");
```

The "chrome" module returns a Components object, which can be unpacked using the destructuring assignment feature provided by Mozilla JavaScript to obtain the Components.* aliases:

1. Cc, otherwise called Components.classes
2. Ci, otherwise called Components.interfaces
3. Cu, otherwise called Components.utils.

4. Cr, otherwise called Components.results.

5. Cm, otherwise called Components.manager.

It is not advisable to use chrome authority in the add-on code unless it is required because the add-ons that uses chrome authority require extra security review before they are made available for distribution to the public.

3.4 Content Scripts

The add-on‘s main code, including "main.js" and other modules in "lib", can use the SDK high-level and low-level APIs, but can‘t access web content directly. Whereas content scripts can‘t use the SDK‘s APIs, but can access web content.

So if we have to build an Add-on that works based on the content of the web page, then we have to make use of the content scripts to access the web page contents. Content scripts are placed in the data subdirectory and they can be loaded into Add-on using contentScript or contentScriptFile option.

Communicating with the add-on:

To enable add-on scripts and content scripts to communicate with each other, each end of the conversation has access to a port object.

1. port.emit() is used to send messages from one side to the other

2. port.on() is used to receive messages sent from the other side

Messages are asynchronous i.e., after sending the message, sender continues processing without waiting for a reply from the recipient.

The add-on code in Figure 28 adds a button to Firefox. When the user clicks

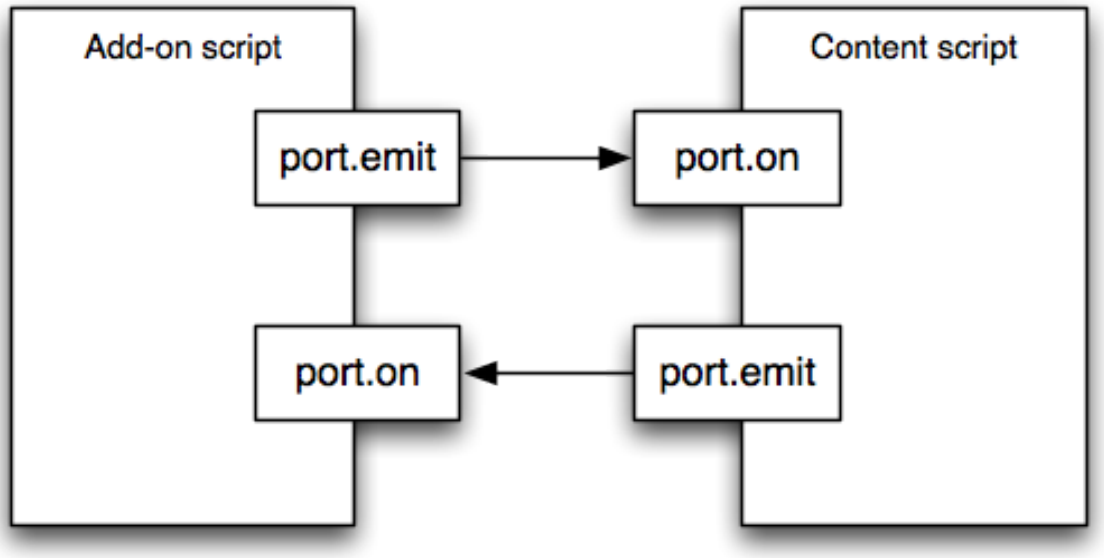

Figure 27: Communication among add-on and content script [1].

this button, add-on attaches a content script to the active tab, sends the **addon-message** to content script. When content script receives add-on message, it will retrieve the first paragraph from the loaded web page and send it to add-on along with **script-response** message. As soon as the add-on receives a response from content script, it logs the first paragraph.

```
//main.js
var tabs = require("sdk/tabs");
var buttons = require("sdk/ui/button/action");
var self = require("sdk/self");

buttons.ActionButton({
  id: "attach-script",
  label: "Attach the script",
  icon: "./icon-16.png",
  onClick: attachScript
});

function attachScript() {
  var worker = tabs.activeTab.attach({
    contentScriptFile: self.data.url("content-script.js")
  });
  worker.port.on("script-response", function(response) {
    console.log(response);
  });
  worker.port.emit("addon-message", "Message from the add-on");
}
```

```
// content-script.js
self.port.on("addon-message", getFirstPara);

function getFirstPara() {
  var paras = document.getElementsByTagName("p");
  if (paras.length > 0) {
    var firstPara = paras[0].textContent;
    self.port.emit("script-response", firstPara);
  }
}
```

Figure 28: Communication among add-on and content script using code [1].

CHAPTER 4

Implementation

This chapter focus on the implementation details of two add-ons:

1. a malicious add-on, which can infect the victim‘s filesystem.
2. a Transcriptase detection add-on, which can detect the metamorphic malware embedded in the web page.

4.1 Malicious add-on

Generally browsers like Firefox, Chrome, and Opera do not allow access to the client filesystem using JavaScript. Even though creating a file is possible in IE using ActiveX objects, the client must enable ActiveX scripts on their system for the ActiveX object related code to execute properly [17].

Firefox add-ons are very powerful because of the high-level APIs that the SDK provides. The SDK has a file I/O module which provides access to the client‘s filesystem.

A malicious add-on was created to demonstrate the way that a victim‘s machine may get infected by a malicious add-on. The basic functionality of this add-on is it provides the statistic value i.e., the total JavaScript bytes in the page loaded by the user, as shown in Figure 29.

The user expects this functionality and installs the malicious add-on, but this add-on also has hidden functionality. Whenever it finds that web page content has Transcriptase in it, it finds all the JavaScript files present in the filesystem and

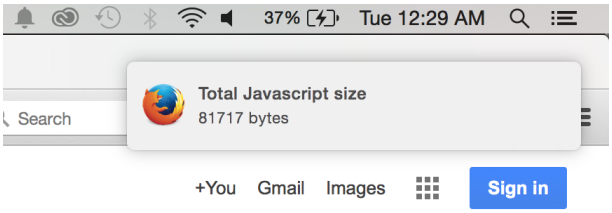

Figure 29: Main Functionality of the malicious add-on

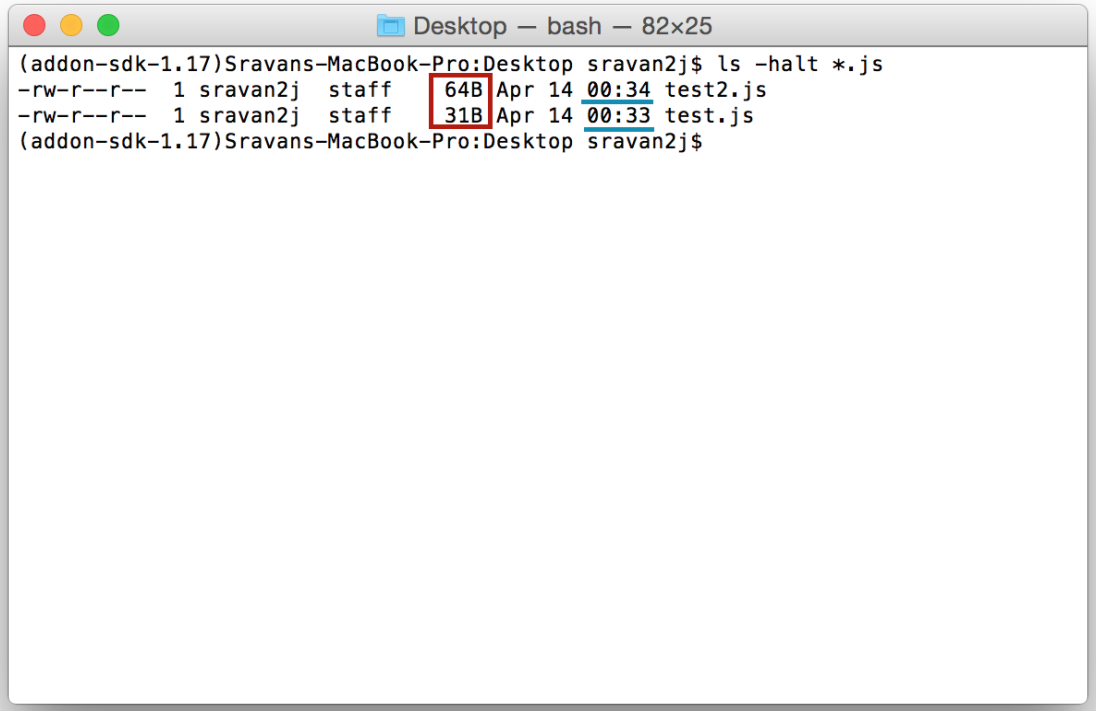

Figure 30: Size of sample JavaScript files before infection

prepends them with Transcriptase code and thus it infects the victim‘s filesystem.

Figure 30 shows the size of our sample JavaScript files before infection and figure 31 shows the size of these files after infection. There is a huge difference in file

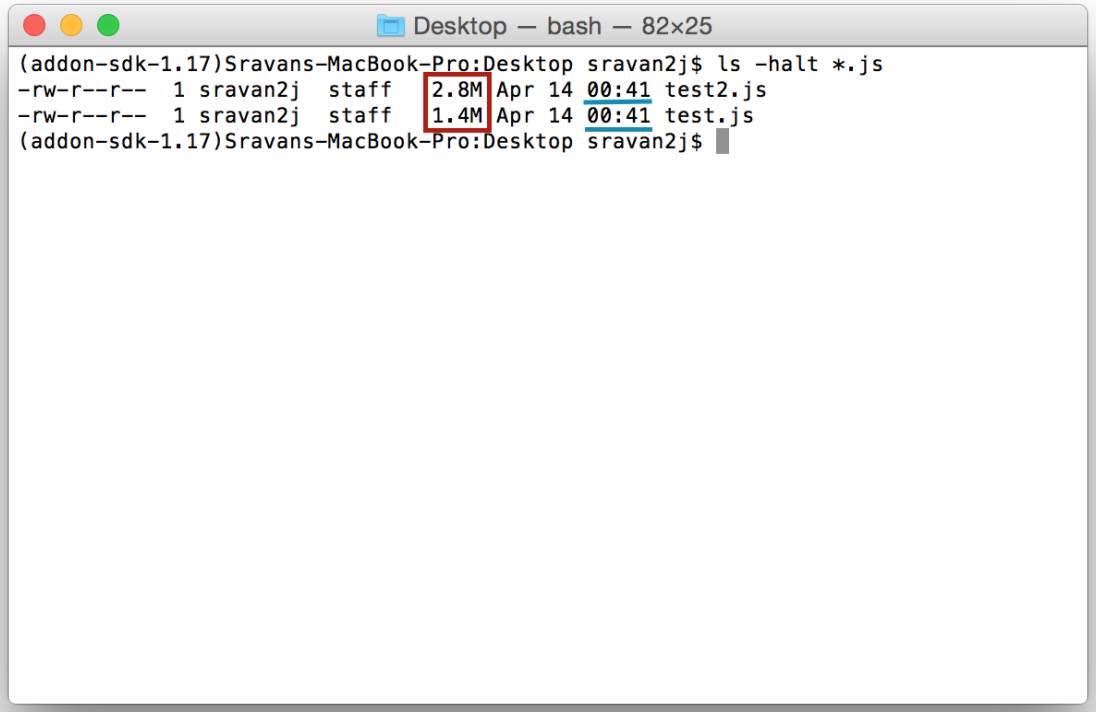

Figure 31: JavaScript file sizes after infection

sizes before and after infection. This infection will remain unknown to the user until the infected files are checked.

4.2 Transcriptase detection add-on

As discussed in Section 2.4, the Transcriptase virus uses different techniques to change its internal structure in order to evade the signature based detection strategy. The Transcriptase detection add-on can detect the Transcriptase malware included in the webpage and notifies the user about the presence of malware without loading the JavaScript malware.

4.2.1 Malware Detection Technique

Despite the fact that metamorphic malware continuously changes its internal structure to stay undetected, still for maintaining its functionality malware places similar instructions (that implements the functionality) somewhere in the code. Thus, all the morphed copies maintain the same statistical distribution of instructions. Different malware detection strategies are designed to make use of these statistical properties like Hidden Markov Models, Opcode Graph Similarity, Simple Substitution Distance, and Singular Value Decomposition.

As mentioned in [4], if the files are having highly similar opcode statistics, then Opcode Graph Similarity and Singular Value Decomposition can classify them better than the Hidden Markov Model and Simple Substitution Distance. Opcode Graph Similarity and Singular Value Decomposition are very sensitive to deadcode, but from the results mentioned in [4] these strategies won‘t be able to distinguish between benign code and virus code only after adding 5000 and 9000 deadcode functions into the virus code respectively. And it is extremely uncommon for a web page to have this much dead code. Adding to this, the ROC curves in [4] shows that Opcode Graph Similarity performs better than Singular Value Decomposition with less than 1000 dead code function insertions.

We used Opcode Graph Similarity technique in the Transcriptase detection add-on.

4.2.2 Opcode Graph Similarity Technique

In [2], Anderson introduced a malware detection technique which is based on analysis of graphs that are constructed using the opcodes of the malware code and test code. In this technique, initially opcodes are extracted from the malware code

and a weighted directed graph is built using the sequence of opcodes. Similarly, a graph is built for the code to be tested. The Manhattan distance between these two weighted graphs specifies the test file score [4].

4.2.3 Opcode Graph

A weighted directed graph built using the sequence of opcodes is known as the 'Opcode Graph'. Each node of this graph specifies a distinct opcode in opcode sequence. A directed edge exists from $node_A$ to $node_B$, if $node_B$'s opcode follows the $node_A$'s opcode in the opcode sequence. The weight of the edge from $node_A$ to $node_B$ specifies the total number of times that $node_B$'s opcode follows $node_A$'s opcode in the entire code.

```
1  PUSH
2  MOV
3  SUB
4  AND
5  MOV
6  TEST
7  JZ
8  INT
9  MOVZX
10 AND
11 MOV
12 MOVZX
13 OR
14 MOV
15 MOV
16 CALL
17 LEAVE
18 RETN
19 ALIGN
20 PUSH
21 MOV
22 MOV
23 PUSH
24 PUSH
25 SUB
26 MOV
27 MOV
28 CALL
29 AND
30 SUB
31 CALL
32 MOV
33 CALL
34 MOV
35 XOR
36 MOV
37 MOV
38 MOV
39 CALL
40 MOV
```

Figure 32: Sample opcode sequence.

Figure 32 shows the sample opcode sequence. The adjacency matrix in

	ALIGN	AND	CALL	INT	JZ	LEAVE	MOV	MOVZX	OR	PUSH	RETN	SUB	TEST	XOR
ALIGN	0	0	0	0	0	0	0	0	0	1	0	0	0	0
AND	0	0	0	0	0	0	2	0	0	0	0	1	0	0
CALL	0	1	0	0	0	1	3	0	0	0	0	0	0	0
INT	0	0	0	0	0	0	0	1	0	0	0	0	0	0
JZ	0	0	0	1	0	0	0	0	0	0	0	0	0	0
LEAVE	0	0	0	0	0	0	0	0	0	0	1	0	0	0
MOV	0	0	4	0	0	0	5	1	0	1	0	1	1	1
MOVZX	0	1	0	0	0	0	0	0	1	0	0	0	0	0
OR	0	0	0	0	0	0	1	0	0	0	0	0	0	0
PUSH	0	0	0	0	0	0	2	0	0	1	0	1	0	0
RETN	1	0	0	0	0	0	0	0	0	0	0	0	0	0
SUB	0	1	1	0	0	0	1	0	0	0	0	0	0	0
TEST	0	0	0	0	1	0	0	0	0	0	0	0	0	0
XOR	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Figure 33: Weight counts adjacency matrix for opcodes in Figure 32.

Figure 33 specifies the weights of the edges formed between these opcodes. For instance, we can find that intersection entry between the CALL row and MOV column has a weight value of 3, which means there are three occurrences of a MOV instruction immediately followed by a CALL instruction in the opcode sequence i.e., at line numbers 15, 27, and 32 in Figure 32.

All the weight counts in Figure 33 are converted into probability values by dividing each row entry by the corresponding row sum. Figure 34 shows the weight probabilities for the Figure 33. Each weight probability specifies the probability of occurrence of a particular opcode, immediately after the selected opcode [4].

Figure 35 shows the opcode graph for the probability matrix in Figure 34

	ALIGN	AND	CALL	INT	JZ	LEAVE	MOV	MOVZX	OR	PUSH	RETN	SUB	TEST	XOR
ALIGN	0	0	0	0	0	0	0	0	0	$^1/_1$	0	0	0	0
AND	0	0	0	0	0	0	$^2/_3$	0	0	0	0	$^1/_3$	0	0
CALL	0	$^1/_5$	0	0	0	$^1/_5$	$^3/_5$	0	0	0	0	0	0	0
INT	0	0	0	0	0	0	0	$^1/_1$	0	0	0	0	0	0
JZ	0	0	0	$^1/_1$	0	0	0	0	0	0	0	0	0	0
LEAVE	0	0	0	0	0	0	0	0	0	0	$^1/_1$	0	0	0
MOV	0	0	$^4/_{14}$	0	0	0	$^5/_{14}$	$^1/_{14}$	0	$^1/_{14}$	0	$^1/_{14}$	$^1/_{14}$	$^1/_{14}$
MOVZX	0	$^1/_2$	0	0	0	0	0	0	$^1/_2$	0	0	0	0	0
OR	0	0	0	0	0	0	$^1/_1$	0	0	0	0	0	0	0
PUSH	0	0	0	0	0	0	$^2/_4$	0	0	$^1/_4$	0	$^1/_4$	0	0
RETN	$^1/_1$	0	0	0	0	0	0	0	0	0	0	0	0	0
SUB	0	$^1/_3$	$^1/_3$	0	0	0	$^1/_3$	0	0	0	0	0	0	0
TEST	0	0	0	0	$^1/_1$	0	0	0	0	0	0	0	0	0
XOR	0	0	0	0	0	0	$^1/_1$	0	0	0	0	0	0	0

Figure 34: Probability matrix for weights adjacency matrix in Figure 33.

4.2.4 Similarity Score Calculation

After creating probability matrices for the malware file and the test file, similarity between two files is calculated by taking the Manhattan distance between two probability matrices. Consider A as the probability matrix of file 1 and each element in A is denoted as $A_{i,j}$ where i and j specifies i^{th} row and j^{th} column respectively. Similarly B is the probability matrix of file 2 and each element in B is denoted as $B_{i,j}$. Similarity between matrix A and B, is calculated as below [8],

Similarity score = $\frac{1}{N^2}(\sum_{i,j=0}^{N-1} |a_{i,j} - b_{i,j}|^2)$

where N is total number of distinct opcodes present in the combination of both files.

Before using the similarity score, we have to determine the threshold score which distinguishes between benign files and malware files. The threshold value is

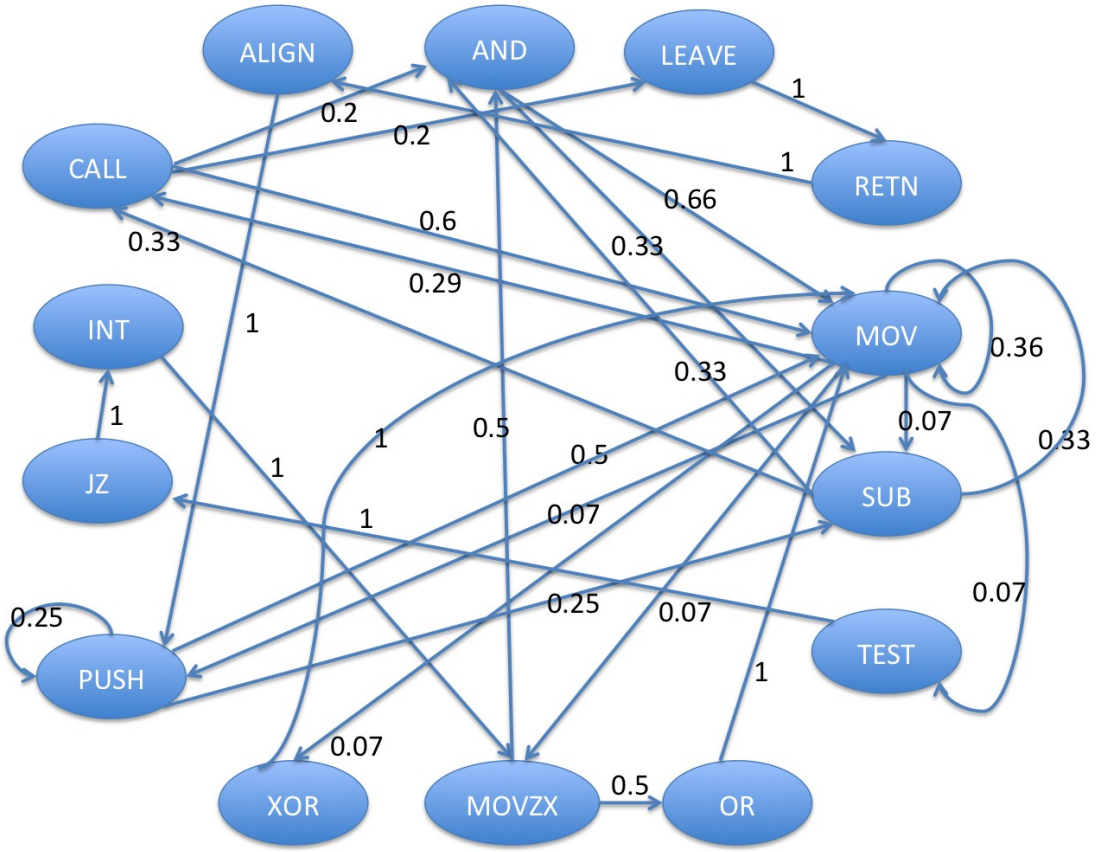

Figure 35: opcode graph for the probability matrix in Figure [8].

determined as follows [8],

1. Construct opcode graphs for all the variants of metamorphic malwares.

2. Construct opcode graphs for all the benign files.

3. Calculate the similarity scores for all pairs of malwares.

4. Calculate the similarity scores for every benign file against malware from step 1.

5. Determine a threshold value using the scores calculated in steps 3 and 4.

4.3 Transcriptase detection add-on architecture

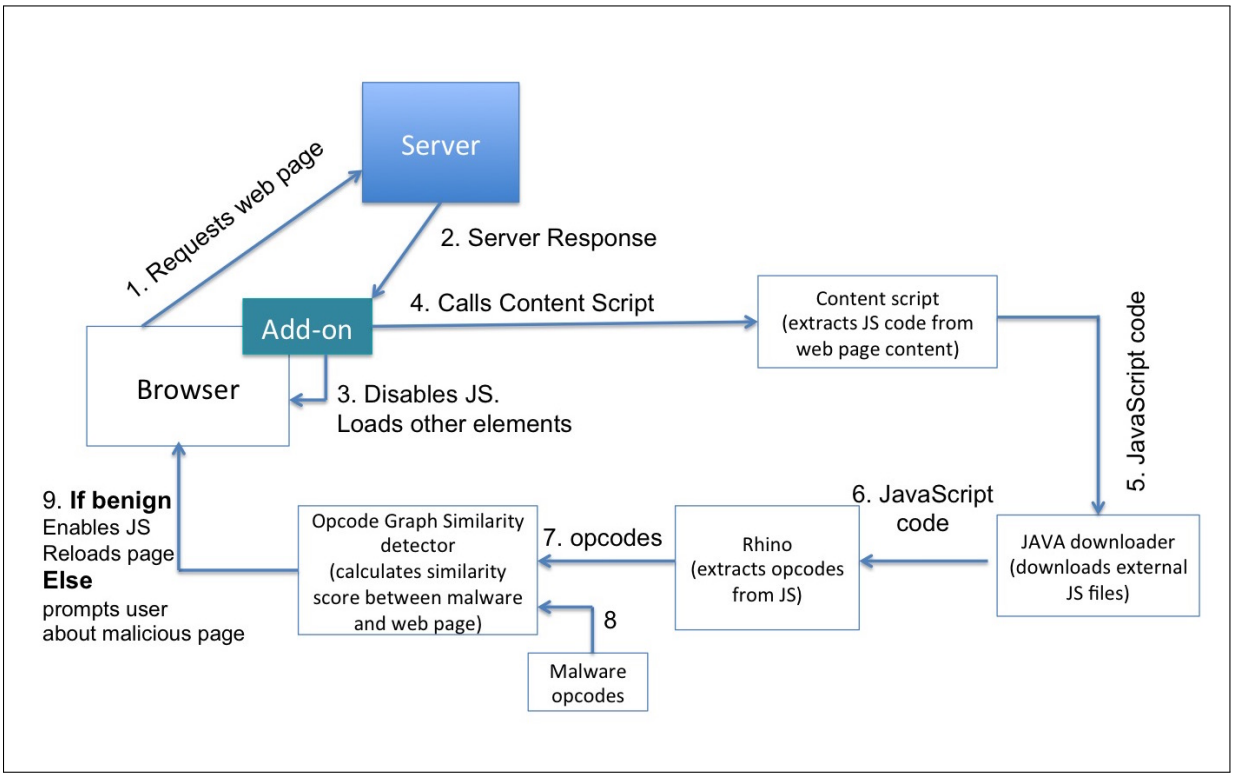

Figure 36: Detection add-on architecture

Figure 36 shows the detection add-on architecture. Each component in the architecture is made up of one or more files. Figure 37 depicts the directory structure of detection add-on.

1. "content-script.js" uses jQuery code to find the JavaScipt content in web page, so included "jquery-1.10.0.min.js" file in the "/data" directory to enable this functionality. content-script.js represents "Content Script" component.

2. The Java files in the add-on are used to download the files, and to calculate the similarity score. All these files are placed in the "/data/java" directory. "Java Downloader" is a combination of DownloadThread, DownloaderApp,

```
Sravans-MacBook-Pro:TranscriptaseDetectionAddon sravan2j$ tree
.
|-- data
|   |-- content-script.js
|   |-- icon-16.png
|   |-- icon-32.png
|   |-- icon-64.png
|   |-- java
|   |   |-- CheckOpcodes.class
|   |   |-- DownloadThread.class
|   |   |-- DownloaderApp.class
|   |   |-- GetScore.class
|   |   |-- ImageDownloader.class
|   |   |-- Lock.class
|   |-- javaext.js
|   |-- jquery-1.10.0.min.js
|   |-- js.jar
|   |-- malware_opcodes.txt
|   |-- opcodes.bat
|   |-- opcodes.sh
|-- lib
|   |-- main.js
|-- package.json
|-- test
    |-- test-main.js

4 directories, 17 files
```

Figure 37: Transcriptase detection add-on directory structure

ImageDownloader, and Lock class files. CheckOpcodes and GetScore class files represents "opcode graph similarity detector".

3. "js.jar" helps the add-on to use the Rhino JS engine.

4. "opcode.sh" is a shell script to invoke Java files. As shell scripts won‘t work on Windows, the "opcode.bat" file is included to invoke Java files on Windows.

5. The "malware_opcodes.txt" file specifies "Malware opcodes" component. This

file contains the opcodes of Transcriptase family malware which is required to validate web page JS content using the opcode graph similarity technique.

More details about these files and the architecture components are covered in subsequent sections.

4.4 JavaScript extraction from web page

As soon as the user enters a web page link, the browser loads the JavaScript content along with HTML and CSS on the page. Before extracting the JS from web page, we have to disable JS load in the browser to prevent the execution of JS malware code.

Enabling/disabling JS feature deals with browser preferences. The preferences system of Mozilla browser can be accessed using XPCOM interfaces like $nsIPrefService$ and $nsIPrefBranch$. The below code is used to disable JavaScript code,

```
1 var prefSrv = this.prefService = Cc["@mozilla.org/preferences-service;1"]
2               .getService(Ci.nsIPrefService);
3 var PBI = Ci.nsIPrefBranch2;
4 this.mozJSPref = prefSrv.getBranch("javascript.").QueryInterface(PBI);
5 this.mozJSPref.setBoolPref("enabled", false);
```

Figure 38: Add-on code to disable JavaScript load on web page.

The code in Figure 38 is interpreted as follows: Line 1 in the above code retrieves the preference services of Mozilla. nsIPrefBranch2 interface, in line 2, allows the add-on to listen to the changes to preferences. Line 3 retrieves the "javascript" preference and queues nsIPrefBranch2 interface using QueryInterface(). Later, setBoolPref() method is used to disable the JS by setting "false" to

"enabled". Similarly the below line of code enables JS load,

```
this.mozJSPref.setBoolPref("enabled", true);
```

After disabling JavaScript, as explained in Section 3.4, the content script i.e., "content-script.js", is used to extract JS. "content-script.js" uses jQuery element selector to find all the <script> elements and extract the JavaScript instructions contained in <script> tags as shown below:

```
var code="";
$("script").each(function(){
    code=code+$(this).html();
});
```

Sometimes, JavaScript code is also placed in an external file and the location of the external JavaScript file is specified in the web page using a src attribute of a <script> element as shown below:

```
<script src="external_javascript.js"></script>
```

There is a chance that this external files may contain JavaScript malware code, so using the code in Figure 39, all the external file's URLs are extracted from the web page. Following is the explanation of the code snippet in Figure 39:

1. "window.location.protocol" returns the protocol of the current web page URL along with colon(:). For instance, "http:", "https:", "ftp:".
   "window.location.host" returns the host name of the web page. For instance, the hostname of
   "http://www.somewebsite.com/tryit.jsp?filename=sample_code" is

```
1  var baseUrl = window.location.protocol + "//" + window.location.host + "/";
2  var Urls = "";
3  var regex = new RegExp("^(?:[a-z]+:)?//", "i");
4  $("script[src]").each(function(){
5    var sourceurl = $(this).attr("src");
6    if(!regex.test(sourceurl))
7    {
8      Urls=Urls+baseUrl;
9    }
10   Urls=Urls+(sourceurl.replace(/^\/+/, ""))+"\n";
11 });
```

Figure 39: JavaScript to extract all the external script file URLs

"www.somewebsite.com". So, line 1 creates a base URL of the web page.

2. line 4 uses a jQuery element selector to retrieve the external file locations defined in src attribute of <script> tag.

3. line 6 uses regex to test whether the external files location is relative or absolute path.

4. Line 8 contains the logic for prepending the base URL to an external file location, if the external file location is relative path.

5. Finally, the "Urls" variable contains all the external local URLs, and these URLs will be saved in a temporary file.

Later, the add-on invokes the *opcodes.sh* file as shown in Figure 41, which performs the following two functions:

Creates a temporary file

"TmpD" returns the temporary directory location of the OS. In line 1, "opcodes.tmp" filename is concatenated to temporary directory path and

getFile() method returns a nsIFile object referring to "<TMP_DIR>/opcodes.tmp" location. Then createUnique() method creates the requested temporary file.

Invokes opcode.sh

The nsIProcess interface is used to execute a process. nsIProcess requires executable name to execute and if the executable file requires any parameters then these parameters need to be passed as args[] to the nsIProcess.

1. Line 3 and 4 creates a nsIFile object referring to executable "/bin/sh".
2. Line 5 and 6 creates an instance of process and initializes it to "/bin/sh" executable.
3. Line 7 and 8 adds both "opcode.sh" file path and temporary file path to "args" array. Then the process is executed using run() which executes the below command internally,

```
$ /bin/sh
    /Users/sravan2j/Downloads/TranscriptaseDetectionAddon/data/opcode.sh
    /tmp/opcodes.tmp
```

Figure 40: command that invokes opcode.sh internally

4.5 Purpose of the Shell script

The opcodes.sh code, shown in Figure 42, performs the following three functions:

1. Line 1 executes following Java files - *DownloadThread.class*, *DownloaderApp.class*, *ImageDownloader.class*, and *Lock.class*, to download

```
Cu.import("resource://gre/modules/FileUtils.jsm");

// create a temporary file
var file = FileUtils.getFile("TmpD", ["opcodes.tmp"]);
file.createUnique(Ci.nsIFile.NORMAL_FILE_TYPE, 0600);

var file = Cc["@mozilla.org/file/local;1"].
          createInstance(Ci.nsILocalFile);
file.initWithPath("/bin/sh");

var process = Cc["@mozilla.org/process/util;1"]
             .createInstance(Ci.nsIProcess);
process.init(file);

var args =
    ["/Users/sravan2j/Downloads/TranscriptaseDetectionAddon/data/opcode.sh"];
// append temporary file path to parameters
args.push(tmpFile.path);
process.run(true, args, args.length);
```

Figure 41: add-on code that creates temporary file and invokes opcode.sh with temporary file

all the external scripts. This java files uses multi threading approach to download all the external scripts in parallel which reduces the total download time.

2. After the above step, the entire JavaScript content will be saved in the /tmp/JSStatements.js file. Line 2 takes the JSStatements.js file as input and generates opcodes for the JavaScript code in JSStatements.js using the Rhino JS engine. The output of this step is the "/tmp/opcodes.txt" file.

3. Line 3 executes CheckOpcodes java code which calculates the similarity score using the opcode similarity technique, between malware_opcodes.txt and the opcodes.txt file. The output of this step is redirected to "$1", which refers to the arguments passed to opcodes.sh. The bash command in Figure 40 shows

that the /tmp/opcodes.tmp file is passed as an argument while calling opcode.sh.

```
java -Xmx500m -cp "." data/DownloaderApp data/externalUrls.txt
java -cp "./data/js.jar" org.mozilla.javascript.tools.jsc.Main
    /tmp/JSStatements.js > /tmp/opcodes.txt
java -cp "./data" CheckOpcodes data/malware_opcodes.txt /tmp/opcodes.txt >
    $1
```

Figure 42: opcode.sh shell script code

4.6 Page validation and clean-up step

The add-on gets the similarity score from the opcodes.tmp file. If the score is less than the threshold value i.e., 0.01, then the web page is considered as a malicious page or else it is a benign page.

1. If the page is benign, then it enables JavaScript and reloads the web page.
2. If the page is malicious, then the web page won't be loaded; instead a prompt is displayed to the user regarding the malware.

At the end, all the temporary files created will be removed as part of the clean-up step.

4.7 Performance improvements

As the add-on performs lot of steps to validate the web page, the execution time will be more. So, instead of validating every web page every time, we can skip the validation during the following scenarios:

4.7.1 Fingerprinting web pages

The hashcode of the benign web page should be saved in the user directory. In the future, when user visits the same web page and if the internal content of the page is not changed from the last visit, then the hash code of the page remains the same as the one that was saved on user‘s machine. In this case, we can safely skip the validation of the web page.

The disadvantage with this approach is that it consumes the user‘s system memory as it saves the hashcode for every web page the user visits.

We can also improve this approach by saving the hash codes in the cloud repository. Whenever any user visits the web page, the add-on connects with the cloud repository and checks if this web page was already validated by any user earlier or not. If it was validated, is the web page hash code the same? And what is the validation result? If the hash code is not in the cloud, then it will be validated by the current user‘s plugin and the result will be stored in the cloud, so that this data will be useful for other users. Because of this approach, the user‘s system memory will be saved and also at any point of time, the web page is validated only once by any user. Necessary security measures should be taken inorder to prevent the attacks like man-in-the-middle attack, cloud data tampering.

4.7.2 Whitelisting websites

Some popular websites are highly secured and regularly monitored, like Google, Facebook, Amazon etc. These websites can be added to benign page list by the user, so that they won‘t be validated by the add-on.

The disadvantage of this approach is that it involves a risk of infection if the

whitelisted web pages are infected by malware.

4.8 Using other detection techniques

Currently this add-on uses only opcode graph similarity detection. Other detection techniques can be used in the add-on by simply changing line 3 of opcode.sh, shown in 42, to execute a program that implements another detection technique instead of executing the CheckOpcodes program. The new program should accept "malware_opcodes.txt" and "/tmp/opcodes.txt" as input files and the similarity score should be saved in the "/tmp/opcodes.tmp" file. No other changes are required.

CHAPTER 5

Testing

To check the accuracy and performance of the add-on, we used malware web pages and benign web pages. To create malware samples, we generated different variants of Transcriptase malware. For benign web pages, we retrieved the JavaScript dead code from http://tools.w3clubs.com/jojo/.

Entire testing is performed on a system with the configuration specified in Table 1:

Table 1: System Specifications

System Model	MacBook Pro (Retina, 13-inch, Mid 2014)
Processor	2.8 GHz Intel Core i5
RAM	16 GB 1600 MHz DDR3
Storage	120 GB
Firefox version	36.0.4
SDK version	Add-on SDK 1.17
Rhino version	Rhino 1.7R4, modified to output opcodes during JS compilation
Java version	1.7.0_71

5.1 Generating Transcriptase variants

Transcriptase was written in JScript, so in a windows system it can be executed by simply double clicking it. From my observation, the generation of each version takes around 15 minutes.

As explained in Section 2.4, Transcriptase carries its source code as meta

instructions and on each execution it creates different variant of its JS source, then prepends that JS code to all the JavaScript files in its directory. So, I followed the below steps to create 100 versions:

1. Created an empty JavaScript file in the Transcriptase directory.
2. Executed Transcriptase, which infects the new empty JavaScript file and converts it to another variant of Transcriptase.
3. Move the older version Transcriptase (or creator Transcriptase) to different folder.
4. Then created an empty JavaScript file in the current folder where the new Transcriptase variant exists.
5. Executed the new variant to infect the empty JavaScript file. Go to Step 3 if the required number of variants aren't generated.

The code in Figure 43 automates the above mentioned steps.

```
FOR %%A IN (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
    25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
    49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
    73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
    97 98 99 100) DO (
transcriptase.js
REN transcriptase.js "var%%~nA.*"
MOVE "var%%~nA.*" "C:\Users\Sravan\Downloads\Transcriptase\versions"
REN empty.js transcriptase.js
COPY "C:\Users\Sravan\Downloads\Transcriptase\template\empty.js" .
)
PAUSE
```

Figure 43: Batch script that automates the Transcriptase variants generations

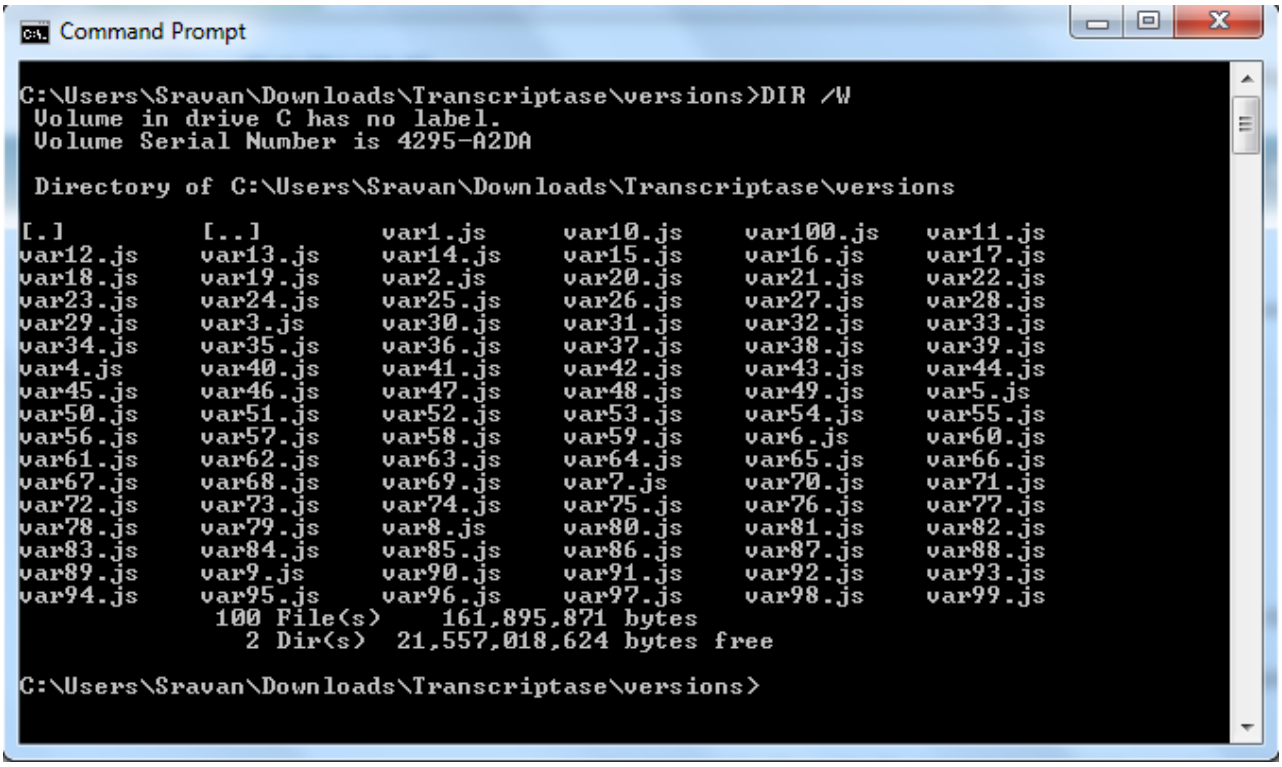

Figure 44: Transcriptase‘s 100 versions.

5.2 Similarity scores and add-on performance

Included console.log() functions in the add-on to log the following details - opcode similarity score and the add-on execution time taken to validate the web page. The below sub section deals with comparison of these details for benign and malware web pages. For testing the add-on, we used 100 samples of benign web page and morphed malware web pages. Malware web pages are morphed by adding randomly generated junk code to it.

5.2.1 Addition of 550 lines of dead code

For this experiment, we used benign web page samples with 550 lines of junk code and also added the same amount of randomly generated junk code to malware

web pages. Tables 2 and 3 contain the details of scores and execution time for the benign and malware samples, respectively. From the table values, we can see that the scores for benign web pages are in the order of 10^{-3} whereas the scores for malware web pages are in the order of 10^{-4}. The graph in Figure 45 clearly shows that the add-on is able to distinguish malware web pages and benign web pages correctly. Only 3 out of 100 malware samples have score similar to benign web pages.

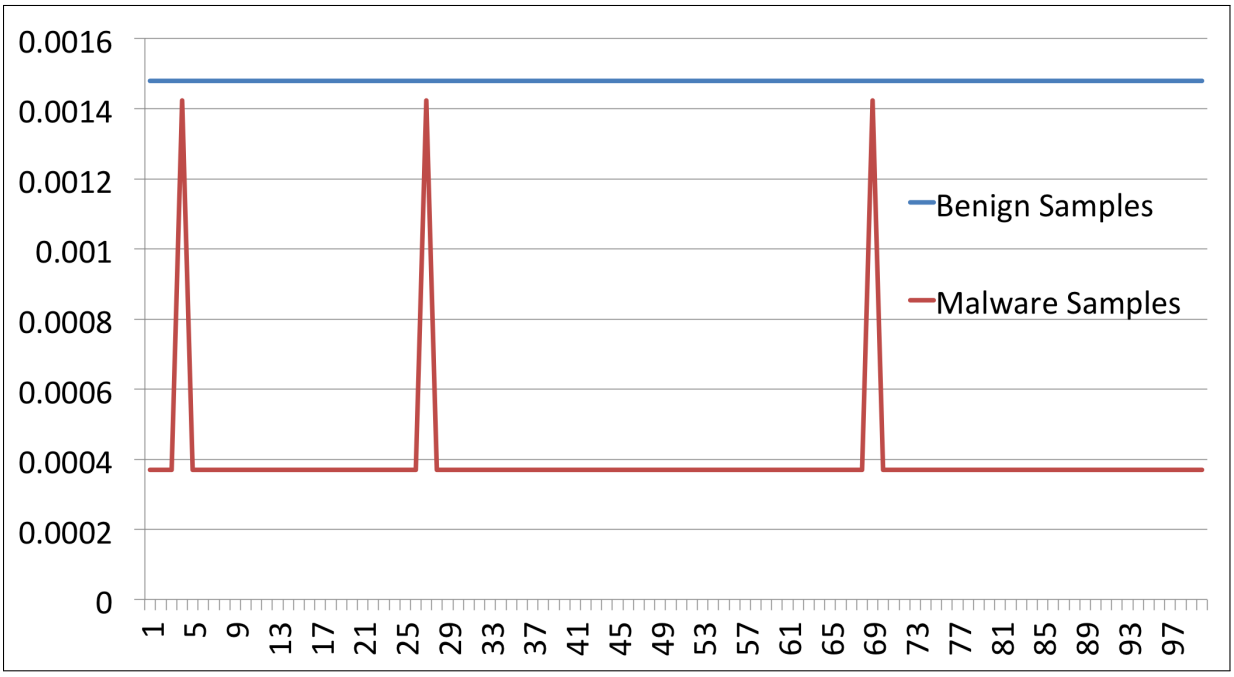

Figure 45: Benign samples scores vs malware samples scores with the addition of 550 lines of dead code

5.2.2 Addition of 5500 lines of dead code

This experiment is same as the above experiment except that here 5500 lines of dead code was included in malware and benign web pages instead of 500 lines. Tables 4 and 5 contains the details of scores and execution time for this experiment. From the table values, we can see that the scores for benign web pages and malware

web pages are still in the order of 10^{-3} and 10^{-4}, respectively. The graph in Figure 46 clearly shows that even after adding 5500 lines of code, the add-on is able to distinguish malware web pages and benign web pages correctly. Only 3 out of 100 malware samples have scores similar to benign web pages.

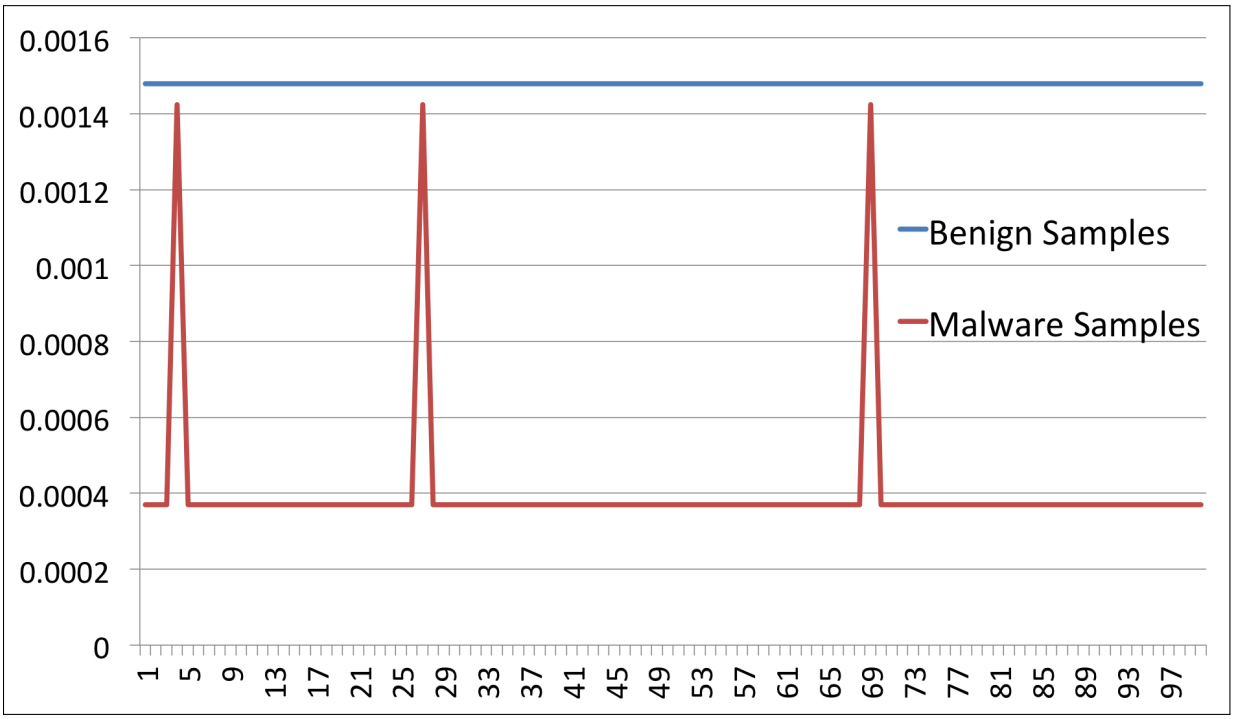

Figure 46: Benign samples scores vs malware samples scores with the addition of 5500 lines of dead code

5.2.3 Addition of 15000 lines of dead code

Here, 15000 lines of dead code were included in malware and benign samples. From the tables 6 and 7, it is clear that scores of malware web pages are varied by a very negligible value when compared to previous experiments. So as shown in Figure 47, the add-on is still able to distinguish malware web pages and benign web pages correctly.

As mentioned in Section 4.2.4, the scores of malware files and benign files can

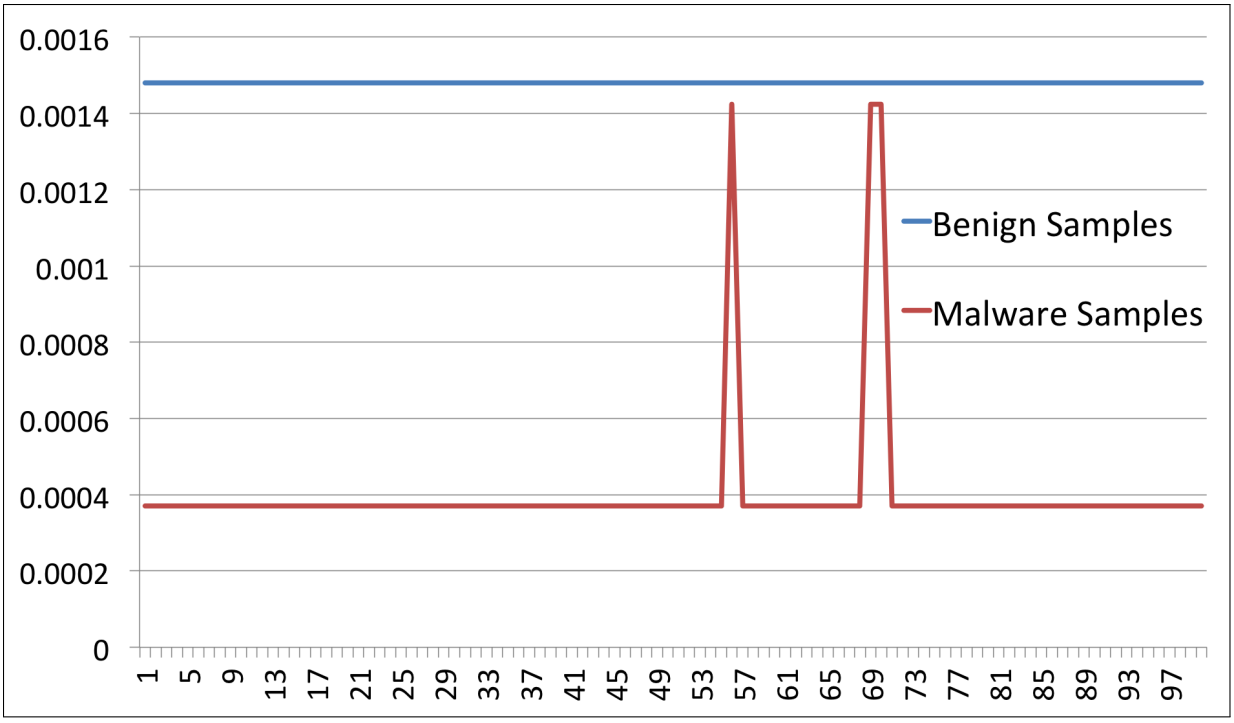

Figure 47: Benign samples scores vs malware samples scores with the addition of 15000 lines of dead code

be compared to calculate the threshold score value of the opcode similarity technique. From the tables 2, 3, 4, 5, 6, and 7, except 9 malwares all other malware scores are in between 0.000369822260 and 0.000369822750. The 9 exception malware scores are between 0.001423994000 and 0.001423994700. And the benign web page scores are between 0.001479288900 and 0.001479291400. The same information is represented with table 8

The threshold value can be any value between 0.00142 and 0.00147. If the lower percentage of false negative rate is acceptable, then the threshold score can be chosen between 0.00036 and 0.00147. In case of malware detection, it is always better to have fewer false negatives, so I use **0.00145** as the threshold score value for the add-on.

5.3 Test for False Positive rate of the add-on

I tested the add-on on popular web sites to detect the "false positive" rate of the add-on. The popular web site links are retrieved from [34].

The table 9 contains the scores and execution time details. All the scores in the table are more than the chosen threshold value (i.e., 0.00145), which means that the add-on validated all the web pages as benign web pages i.e., add-on has zero false positive rate.

5.4 Splitting Transcriptase code

Transcriptase can be split into several external JS files and then the external files can be included in a web page. So, the following experiment was performed to calculate the scores of split files by dividing Transcriptase into a various number of files.

The code was split based on functions count. The experiment was started by dividing the Transcriptase code into two files with almost equal number of functions and then continued till the split files count reaches 76. A parser was developed in Python to detect the valid start and end point of the JavaScript functions and to properly split the Transcriptase code. Thus the resultant split files are syntactically correct. Corresponding parser code is shown in Appendix A, Section A.1.

Table 10 shows the results for various splits. The "Max" and "Min" column specify the maximum and minimum similarity score among the split files, respectively. The "count" column specifies the number of files the Transcriptase code was split into.

Figure 48 is a graphical representation of the Table 10 values. The graph

clearly shows that even when the code was split, **the minimum score among the split files is always less than threshold which means that there always exists at least one split file with score a less than threshold score**. Thus, it is possible to detect the malware even by testing all the external files separately. So, we can validate all the external scripts parallely to increase the performance of the add-on.

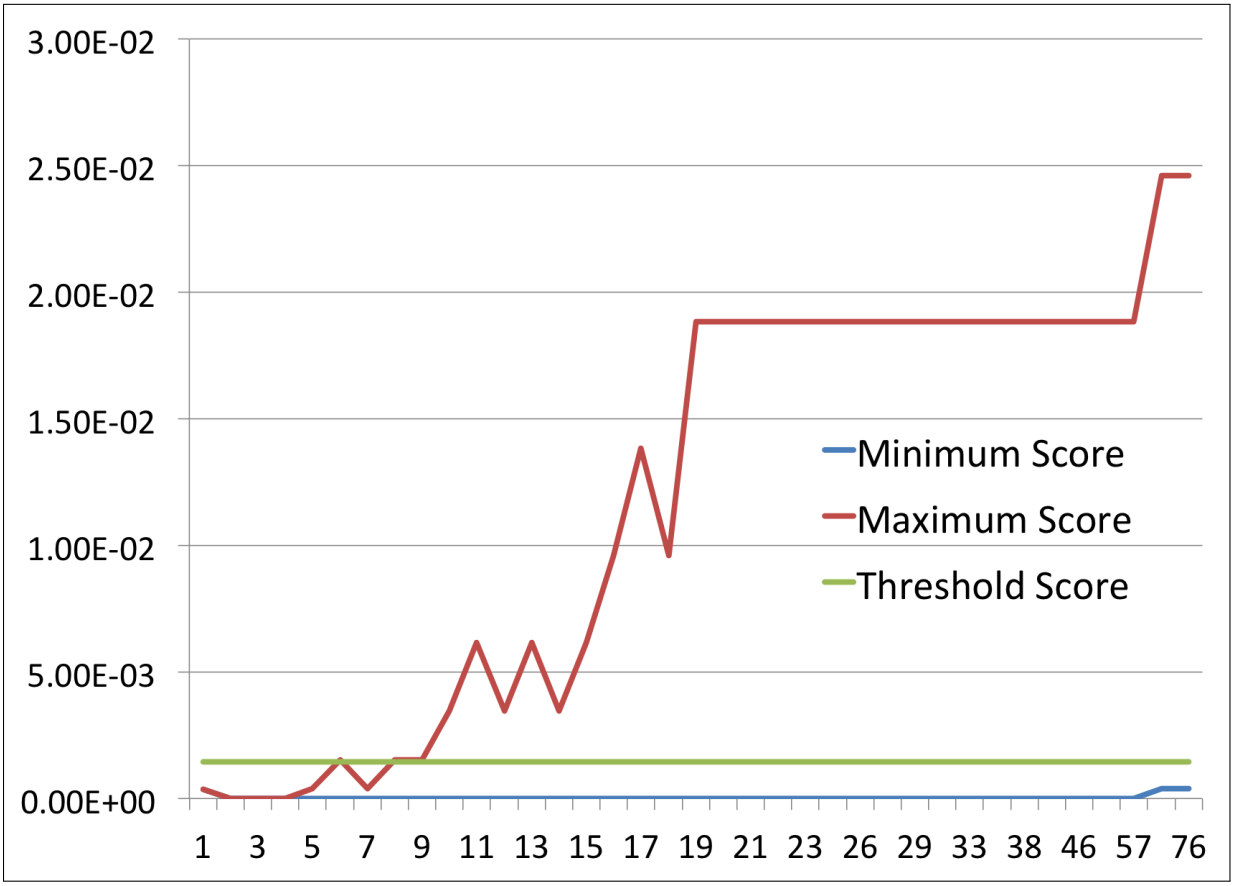

Figure 48: Graph showing min and max scores of Transcriptase split files.

Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)
0.00147929	1136	0.00147929	1135	0.00147929	1482	0.0014792903	1232
0.00147929	1172	0.0014792896	1124	0.0014792896	1304	0.0014792907	1115
0.00147929	1150	0.0014792907	1113	0.00147929	1245	0.00147929	1308
0.00147929	1125	0.0014792903	1178	0.0014792907	1274	0.0014792896	1159
0.0014792893	1080	0.0014792896	1133	0.0014792896	1235	0.00147929	1125
0.00147929	1223	0.0014792903	1133	0.0014792903	1242	0.0014792903	1228
0.0014792896	1125	0.00147929	1134	0.0014792903	1246	0.0014792896	1180
0.0014792903	1132	0.0014792893	1106	0.0014792903	1250	0.0014792893	1131
0.001479291	1120	0.00147929	1142	0.0014792903	1257	0.00147929	1136
0.0014792903	1198	0.0014792896	1159	0.0014792903	1255	0.0014792907	1132
0.0014792903	1134	0.0014792896	1125	0.0014792903	1214	0.0014792896	1141
0.0014792907	1131	0.0014792907	1122	0.0014792903	1152	0.00147929	1134
0.0014792903	1108	0.00147929	1176	0.0014792896	1116	0.0014792896	1124
0.00147929	1110	0.0014792903	1134	0.0014792907	1138	0.0014792903	1410
0.00147929	1124	0.0014792893	1113	0.00147929	1105	0.0014792907	1123
0.0014792893	1122	0.00147929	1168	0.0014792907	1093	0.00147929	1150
0.0014792907	1116	0.00147929	1135	0.0014792903	1175	0.0014792903	1175
0.0014792896	1117	0.0014792907	1118	0.0014792907	1121	0.0014792903	1116
0.0014792903	1198	0.0014792903	1148	0.0014792903	1127	0.0014792893	1125
0.00147929	1122	0.0014792907	1102	0.0014792903	1167	0.0014792896	1143
0.00147929	1137	0.0014792893	1144	0.0014792903	1105	0.0014792903	1223
0.00147929	1204	0.0014792896	1098	0.0014792907	1113	0.0014792903	1120
0.0014792907	1130	0.00147929	1152	0.0014792896	1111	0.0014792907	1140
0.00147929	1104	0.0014792903	1129	0.00147929	1140	0.00147929	1213
0.0014792903	1119	0.00147929	1215	0.00147929	1149	0.00147929	1119

Table 2: Table illustrating the scores and add-on execution time for 100 benign web pages, in four columns (i.e., 25 samples per column). Benign webpages are generated with 550 lines of dead code.

Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)
0.00036982258	7897	0.00036982243	6707	0.00036982258	7888	0.0003698225	6642
0.00036982258	6875	0.001423994	7493	0.00036982243	8133	0.00036982234	7355
0.00036982258	6284	0.0003698224	6041	0.0003698224	8576	0.00036982235	7489
0.0014239943	6430	0.00036982266	7119	0.00036982258	7785	0.00036982266	7072
0.00036982234	7629	0.0003698225	7223	0.00036982243	8536	0.00036982258	6941
0.00036982238	7682	0.00036982258	7159	0.00036982234	7118	0.0003698225	7372
0.00036982266	7774	0.0003698225	8297	0.0003698225	7138	0.00036982243	7832
0.00036982266	8075	0.00036982243	6949	0.00036982243	6917	0.00036982258	7158
0.0003698225	6127	0.00036982243	8550	0.00036982243	6475	0.00036982258	6603
0.00036982258	7312	0.0003698225	7157	0.00036982243	8109	0.0003698224	8482
0.00036982258	8038	0.0003698225	7532	0.00036982258	7013	0.00036982243	7718
0.00036982258	10123	0.0003698224	6904	0.00036982258	6740	0.00036982258	6723
0.00036982236	8091	0.0003698225	6679	0.0003698225	7239	0.00036982258	8046
0.00036982245	6392	0.00036982258	7031	0.0003698225	7138	0.0003698225	7845
0.00036982238	6518	0.00036982258	6438	0.00036982258	6088	0.00036982258	7862
0.00036982251	7121	0.00036982266	8230	0.00036982243	7529	0.00036982258	9817
0.00036982243	6309	0.0003698225	9986	0.0003698224	8031	0.0003698224	7804
0.00036982258	8646	0.0003698225	6958	0.0003698224	7753	0.0003698225	9849
0.00036982243	6706	0.00036982243	10166	0.0014239943	6494	0.0003698225	9302
0.00036982258	6617	0.0003698225	8358	0.00036982258	6815	0.00036982256	7239
0.00036982241	6502	0.00036982251	6129	0.00036982266	8174	0.00036982263	7652
0.00036982232	7289	0.00036982249	6732	0.00036982275	8314	0.0003698225	7261
0.0003698224	7236	0.00036982258	7730	0.00036982266	7465	0.00036982258	6398
0.00036982258	7746	0.00036982258	7719	0.00036982266	7113	0.0003698225	8812
0.00036982243	7267	0.00036982258	7992	0.00036982258	6757	0.00036982243	8357

Table 3: Table illustrating the scores and add-on execution time for 100 malware web pages, in four columns (i.e., 25 samples per column). Malware webpages are morphed with 550 lines of dead code.

Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)
0.0014792907	2287	0.0014792903	1937	0.00147929	2379	0.0014792903	3148
0.00147929	2447	0.0014792893	2813	0.0014792896	1966	0.0014792907	2210
0.00147929	2215	0.0014792903	2153	0.0014792903	2728	0.0014792903	2094
0.0014792907	2376	0.0014792903	2604	0.00147929	2167	0.0014792903	2068
0.00147929	2188	0.0014792914	2221	0.0014792903	2022	0.0014792907	2462
0.0014792903	2661	0.00147929	2152	0.0014792907	2383	0.0014792893	2060
0.0014792907	1989	0.0014792903	2404	0.0014792903	2033	0.00147929	2074
0.00147929	2137	0.00147929	2260	0.00147929	2463	0.0014792903	2254
0.0014792903	2001	0.0014792907	2110	0.0014792903	2535	0.0014792893	2564
0.0014792907	2003	0.0014792893	2226	0.0014792903	2026	0.0014792903	2084
0.00147929	2646	0.0014792903	2134	0.0014792907	2030	0.0014792903	2406
0.0014792903	1969	0.0014792903	2115	0.0014792907	1944	0.0014792903	2437
0.0014792903	1972	0.0014792903	2003	0.0014792893	2239	0.0014792889	2317
0.0014792903	1937	0.0014792893	1989	0.0014792893	2270	0.0014792903	2001
0.0014792903	1992	0.00147929	2599	0.00147929	1954	0.0014792903	1933
0.0014792896	2612	0.0014792893	2277	0.0014792907	1961	0.00147929	1994
0.0014792903	2092	0.0014792896	1971	0.0014792893	2240	0.0014792903	1967
0.0014792907	2453	0.0014792903	2388	0.0014792907	2280	0.00147929	2279
0.0014792903	2028	0.00147929	1983	0.00147929	1966	0.0014792903	1968
0.00147929	2018	0.0014792903	1996	0.0014792903	1985	0.001479291	1978
0.0014792903	2010	0.0014792903	1990	0.00147929	1978	0.0014792907	1946
0.0014792896	1981	0.0014792903	2034	0.00147929	1995	0.0014792903	1980
0.0014792893	1967	0.00147929	1980	0.0014792907	1987	0.00147929	1942
0.0014792903	1938	0.0014792903	2005	0.00147929	1962	0.00147929	1989
0.0014792896	2324	0.0014792903	2103	0.0014792907	2497	0.0014792903	1985

Table 4: Table illustrating the scores and add-on execution time for 100 benign web pages, in four columns (i.e., 25 samples per column). Benign webpages are generated with 5500 lines of dead code.

Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)
0.00036982243	8233	0.00036982243	8232	0.0003698225	8499	0.00036982258	7472
0.0003698225	7442	0.0014239947	8681	0.0003698225	7973	0.00036982226	8276
0.0003698225	7297	0.0003698225	6720	0.0003698224	8310	0.00036982235	8916
0.0014239941	7827	0.0003698225	7818	0.00036982258	7843	0.00036982243	7925
0.00036982258	7850	0.00036982258	8466	0.00036982258	7476	0.00036982258	7405
0.00036982247	8132	0.00036982243	7717	0.00036982234	7933	0.0003698224	8014
0.00036982258	8478	0.0003698225	9749	0.00036982258	7844	0.0003698225	8181
0.0003698225	8910	0.00036982243	7992	0.00036982234	9089	0.0003698225	7358
0.0003698224	7029	0.0003698225	7495	0.00036982243	7412	0.00036982266	7505
0.00036982234	8039	0.00036982258	7474	0.0003698225	9174	0.00036982275	8958
0.00036982234	7937	0.00036982243	7743	0.0003698225	7246	0.00036982258	8510
0.00036982258	8008	0.00036982266	7970	0.0003698225	7440	0.0003698225	7740
0.00036982243	8012	0.00036982266	8106	0.00036982258	8472	0.00036982258	8390
0.00036982257	7884	0.00036982258	7892	0.0003698224	7832	0.00036982258	8352
0.00036982241	8593	0.0003698225	7070	0.0003698225	6720	0.00036982243	8671
0.00036982249	8147	0.0003698225	7188	0.00036982243	8236	0.0003698224	8519
0.0003698225	7143	0.00036982232	8449	0.00036982258	8064	0.0003698225	6830
0.0003698225	8747	0.00036982266	7731	0.00036982243	8323	0.0003698224	8421
0.00036982258	7226	0.00036982258	8379	0.0014239943	7039	0.00036982243	8282
0.00036982258	7001	0.00036982243	9518	0.00036982251	7213	0.00036982249	7813
0.00036982249	6871	0.00036982249	9287	0.00036982243	9476	0.00036982252	8132
0.00036982243	7892	0.00036982252	8936	0.00036982266	8468	0.0003698224	7930
0.0003698225	7989	0.0003698224	8250	0.00036982243	8304	0.0003698225	7418
0.00036982234	7814	0.0003698224	8658	0.00036982258	7766	0.0003698225	9372
0.0003698225	8370	0.00036982258	8248	0.00036982258	7644	0.00036982266	8933

Table 5: Table illustrating the scores and add-on execution time for 100 malware web pages, in four columns (i.e., 25 samples per column). Malware webpages are morphed with 5500 lines of dead code.

Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)
0.0014792903	3241	0.001479291	3301	0.0014792903	3274	0.0014792903	3484
0.0014792903	3180	0.00147929	3162	0.00147929	3355	0.0014792896	3342
0.00147929	3172	0.0014792896	3134	0.00147929	3345	0.0014792903	3415
0.0014792893	3291	0.0014792903	3150	0.00147929	3185	0.0014792896	3409
0.001479291	3161	0.0014792896	3545	0.0014792907	3184	0.0014792896	3536
0.0014792903	3178	0.00147929	3104	0.00147929	3445	0.00147929	3540
0.0014792907	3182	0.00147929	3167	0.0014792896	4519	0.0014792903	3794
0.00147929	3378	0.0014792896	3307	0.00147929	3448	0.0014792896	3786
0.00147929	3575	0.0014792907	3192	0.0014792907	3361	0.0014792896	3696
0.00147929	3563	0.0014792896	3131	0.00147929	3181	0.0014792896	3513
0.0014792903	3271	0.0014792903	3216	0.0014792903	3383	0.0014792903	3405
0.0014792907	3188	0.0014792889	3320	0.0014792903	4725	0.0014792907	3608
0.0014792903	3122	0.0014792896	3137	0.0014792903	3602	0.00147929	3275
0.00147929	3467	0.0014792903	3225	0.0014792907	3217	0.0014792893	3439
0.0014792903	3197	0.0014792903	3328	0.00147929	3464	0.00147929	3700
0.0014792907	3149	0.0014792903	3329	0.00147929	3159	0.0014792907	3230
0.0014792893	3139	0.00147929	3201	0.0014792903	4870	0.00147929	3221
0.0014792893	3100	0.0014792896	3171	0.0014792896	3244	0.0014792907	3199
0.00147929	3206	0.0014792903	4664	0.00147929	3292	0.0014792896	3239
0.0014792903	3139	0.00147929	3105	0.00147929	3417	0.00147929	3133
0.0014792896	3159	0.0014792893	3177	0.0014792903	3341	0.0014792903	3300
0.001479291	3102	0.00147929	3467	0.00147929	4321	0.0014792903	3420
0.0014792903	3267	0.0014792903	3426	0.00147929	3587	0.0014792896	5329
0.00147929	3359	0.00147929	3421	0.0014792893	3333	0.0014792907	3614
0.0014792903	3412	0.00147929	3400	0.001479291	3289	0.0014792903	3450

Table 6: Table illustrating the scores and add-on execution time for 100 benign web pages, in four columns (i.e., 25 samples per column). Benign webpages are generated with 15000 lines of dead code.

Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)	Score	Time (milliseconds)
0.00036982258	9192	0.0003698224	9179	0.0003698225	9225	0.00036982266	8274
0.00036982258	9105	0.0003698225	7719	0.0003698225	9240	0.00036982258	8962
0.0003698225	9104	0.00036982258	8207	0.0003698225	9458	0.00036982263	9132
0.0003698225	9303	0.00036982243	9510	0.00036982266	8810	0.0003698225	8588
0.00036982258	8568	0.00036982258	9779	0.00036982243	8545	0.0003698224	8185
0.00036982241	9942	0.0003698225	9537	0.0014239943	8158	0.0003698224	8915
0.00036982266	9502	0.00036982266	9766	0.00036982258	8540	0.00036982266	9395
0.0003698224	9791	0.0003698225	8802	0.0003698225	9080	0.00036982258	8165
0.0003698225	7951	0.00036982243	10122	0.00036982258	9233	0.00036982234	8113
0.0003698225	8565	0.0003698225	9180	0.0003698224	8556	0.0003698225	9760
0.0003698225	8059	0.00036982234	8254	0.0003698224	9702	0.0003698224	9330
0.0003698225	8889	0.00036982258	8146	0.00036982258	9082	0.00036982258	8756
0.0003698225	8884	0.00036982258	7851	0.0003698225	9281	0.0003698225	9336
0.00036982235	9126	0.00036982258	8672	0.00036982258	8465	0.0003698225	8828
0.00036982251	9631	0.00036982258	7948	0.00036982258	9770	0.00036982258	9277
0.0003698226	8128	0.00036982266	8689	0.00036982258	9211	0.0003698225	9478
0.00036982243	7919	0.0003698225	8264	0.00036982243	9809	0.00036982258	7847
0.0003698225	9433	0.0003698225	8562	0.0003698224	9109	0.00036982266	9312
0.0003698225	8202	0.00036982258	8119	0.001423994	9592	0.00036982266	8933
0.00036982243	8400	0.00036982266	8845	0.0014239942	1070	0.00036982249	9612
0.00036982252	9783	0.00036982266	8845	0.00036982266	9750	0.00036982258	8495
0.00036982258	8811	0.00036982266	8845	0.00036982243	9469	0.0003698224	8855
0.00036982275	10339	0.0003698225	9078	0.0003698224	8303	0.00036982258	7895
0.0003698225	8925	0.00036982258	9246	0.00036982258	8821	0.0003698224	10477
0.00036982243	9367	0.0003698224	8687	0.00036982258	8830	0.00036982243	8510

Table 7: Table illustrating the scores and add-on execution time for 100 malware web pages, in four columns (i.e., 25 samples per column). Malware web pages are morphed with 15000 lines of dead code.

Table 8: Table illustrating the max and min scores for all the sample files, after comparing the scores from the tables 2, 3, 4, 5, 6, and 7.

	Min Score	Max Score
291 malware samples	0.000369822260	0.000369822750
9 exceptional malware samples	0.001423994000	0.001423994700
300 benign samples	0.001479288900	0.001479291400

Table 9: Table contains scores and add-on execution time details for popular web pages.

Web page	Score	Time (Milli seconds)	Result
https://www.google.com/	0.025195263	2189	Benign page
https://www.facebook.com/	0.08652405	3190	Benign page
http://www.baidu.com/	0.051652893	4442	Benign page
http://www.twitter.com/	0.8132002	4076	Benign page
http://www.taobao.com	0.29001402	3401	Benign page
http://www.qq.com/	0.014076417	3177	Benign page
https://www.linkedin.com	0.0916255	4309	Benign page
https://live.com	0.30142236	1379	Benign page
http://www.sina.com.cn/	0.04421566	2513	Benign page
http://us.weibo.com/gb	0.210642001	2921	Benign page
http://www.hao123.com/	0.046390533	5314	Benign page
http://www.bing.com/	0.30142236	1323	Benign page
http://www.apple.com/	0.0625	5505	Benign page
http://www.aliexpress.com/	0.06497499	1794	Benign page
http://www.imdb.com/	0.041259766	8022	Benign page
http://www.alibaba.com/	0.0047562416	3206	Benign page
http://www.ask.com/	0.051652893	4611	Benign page
https://www.netflix.com	0.0625	10046	Benign page
http://www.naver.com/	0.30142236	2504	Benign page
http://diply.com/	0.05702829	8612	Benign page
https://mail.google.com	0.30142236	1713	Benign page
http://www.youku.com/	0.051652893	4044	Benign page
http://www.flipkart.com/	0.018838914	1604	Benign page
https://www.amazon.com/	0.100754	2839	Benign page
http://www.wikipedia.org/	0.094681033	1914	Benign page

Table 10: Splitting Transcriptase into several files

Count	Min	Max	Count	Min	Max
1	3.70E-04	3.70E-04	20	1.68E-18	0.018838914
2	1.28E-18	1.16E-17	21	1.81E-17	0.018838914
3	1.77E-19	1.39E-17	22	1.69E-19	0.018838914
4	6.17E-19	1.75E-17	23	9.64E-20	0.018838914
5	4.41E-20	3.84E-04	24	7.37E-19	0.018838914
6	5.36E-18	0.00153787	26	1.61E-19	0.01883891
7	4.57E-19	3.84E-04	27	2.27E-17	0.018838914
8	2.19E-18	0.00153787	29	9.72E-19	0.018838914
9	1.86E-17	0.00153787	31	3.85E-18	0.018838914
10	1.69E-19	0.003460208	33	6.35E-18	0.018838914
11	1.73E-18	0.00615148	35	4.33E-18	0.018838914
12	3.91E-18	0.003460208	38	4.37E-18	0.018838914
13	5.64E-20	0.00615148	42	8.12E-18	0.018838914
14	5.47E-19	0.003460208	46	4.41E-20	0.018838914
15	1.88E-18	0.00615148	51	5.21E-18	0.018838914
16	8.01E-20	0.009611688	57	1.96E-17	0.018838914
17	1.26E-18	0.013840835	65	3.84E-04	0.024605926
18	5.07E-19	0.009611691	76	3.84E-04	0.024605926
19	8.01E-19	0.018838914			

CHAPTER 6

Conclusion and Future Enhancements

The aim of this research was to build a Mozilla add-on to detect metamorphic JavaScript malware embedded in a web page. For this purpose, I implemented an add-on using the Mozilla add-on SDK. Internally, the add-on uses the Rhino JavaScript engine to generate opcodes for the JavaScript content of a web page. As the opcode graph similarity technique performs better while classifying the files with similar opcode statistics, this technique was used in the add-on as a malware detection technique. Test results from chapter 5 show that a threshold score value **0.00145** is able to classify the Transcriptase malware family viruses and benign web pages properly even after adding significant amount of junk code. A similar approach can be used for all the different types of metamorphic malware.

Test results also show that execution time for the add-on is around 1 to 4 seconds for benign web pages and 6 to 11 seconds for malware web pages. Even though the execution overhead seems significant, the user is able to view the HTML and CSS content of the page properly during the add-on execution period. As discussed in Section 4.7.1, future enhancements for this thesis can include extending the add-on to use the cloud to increase the add-on performance. This enhancement requires efficient security measures, so that an intruder can't eavesdrop/tamper with the information passed to and from cloud.

Future enhancements also include eliminating the burden of validating some external JavaScript files by storing their links as white lists. For instance, several web pages may have JavaScript code to display Google Ads, as Google is secured

and regularly monitored, we can safely consider all the external Google Ads related JavaScript files as benign files. This approach may also involve some risk if any of the web page in the white list is attacked.

Different malware detection techniques can be added to the add-on to increase the detection rate. As discussed in Section 4.8, the add-on provides simple way to include other detection techniques.

LIST OF REFERENCES

[1] Vold, Erik, Will Bamberg, Kosmodrey, and Aviav. Add-on SDK Tutorials. Mozilla Developer Network. Web. Retrieved 25 Sept. 2014, from https://developer.mozilla.org/en-US/Add-ons/SDK/Tutorials

[2] Anderson, B., Quist, D., Neil, J., Storlie, C., & Lane, T. (2011). Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4), 247-258.

[3] Jetpack. MozillaWiki. Web. Retrieved 16 Dec. 2014, from https://wiki.mozilla.org/Jetpack

[4] Musale, M., Austin, T. H., & Stamp, M. (2014). Hunting for metamorphic JavaScript malware. *Journal of Computer Virology and Hacking Techniques*, 1-14.

[5] Ferrie, Peter. READ THE TRANSCRIPT. Virus Bulletin. (May 2013). https://www.virusbtn.com/pdf/magazine/2013/201305.pdf

[6] Protecting The Reputation Of Your Online Business, StopTheHacker. (Feb. 2012). https://www.stopthehacker.com/wp-content/uploads/2012/02/Protecting-The-Reputation-Of-Your-Online-Business4.pdf

[7] Sanger, David E., and Nicole Perlroth. Bank Hackers Steal Millions via Malware. The New York Times (14 Feb. 2015). Web. Retrieved 17 Feb. 2015, from http://www.nytimes.com/2015/02/15/world/bank-hackers-steal-millions-via-malware.html?_r=0

[8] Runwal, N., Low, R. M., & Stamp, M. (2012). Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, 8(1-2), 37-52

[9] Sophisticated Malware Forecasted to Escalate in 2015, Experts Report. (2 Feb. 2015). Retrieved 24 Feb. 2015, from https://www.allclearid.com/blog/sophisticated-malware-forecasted-to-escalate-in-2015-experts-report

[10] SilverRhino, Kevin G. Coleman. Cybersecurity Is No Longer an Option. C4ISR & Networks. (12 Jan. 2015). Web. Retrieved 7 Feb. 2015, from http://www.c4isrnet.com/story/military-tech/blog/net-defense/2015/01/12/coleman-cybersecurity-imperative/21632093/

[11] Schiffman, Mike. A Brief History of Malware Obfuscation. Cisco Blog RSS. Cisco. (15 Feb. 2010). Web. Retrieved 18 Nov. 2014, from http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2

[12] Runwal, N. (2011). Graph technique for metamorphic virus detection.

[13] Mycroft Project: Search Engine Plugins - Firefox IE Chrome. Web. Retrieved 13 Feb. 2015, from http://mycroftproject.com/

[14] Nachenberg, C. (1996). Understanding and managing polymorphic viruses. The Symantec Enterprise Papers, 30, 16.

[15] Sharma, A., & Sahay, S. (2014). Evolution and detection of polymorphic and metamorphic malwares: A survey. ArXiv Preprint arXiv:1406.7061,

[16] Stamp, M. (2011). *Information security: principles and practice*. John Wiley & Sons.

[17] Creating a File on Client Side Using JavaScript. Stack Overflow. Web. Retrieved 12 Dec. 2014, from http://stackoverflow.com/a/3950151

[18] Li, X., Loh, P. K., & Tan, F. (2011). Mechanisms of polymorphic and metamorphic viruses. *Paper presented at the Intelligence and Security Informatics Conference (EISIC)*, 2011 European, 149-154.

[19] Tanenbaum, Andrew S. Modern Operating Systems. Second ed. Englewood Cliffs, NJ: Prentice Hall, 1992. Print. Chapter 9, pp. 617-637.

[20] Worthman, Ernest. The Next Big Threat: AI Malware. Semiconductor Engineering. (7 July 2014). Web. Retrieved 14 Jan. 2015, from http://semiengineering.com/the-next-big-threat-ai-malware/

[21] Stepan, A. E. (2005). Defeating polymorphism: Beyond emulation. *Paper presented at the Proceedings of the Virus Bulletin International Conference.*

[22] Ször, P., & Ferrie, P. (2001). Hunting for metamorphic. *Paper presented at the Virus Bulletin Conference.*

[23] Desai, P., & Stamp, M. (2010). A highly metamorphic virus generator. *International Journal of Multimedia Intelligence and Security*, 1(4), 402-427.

[24] Orr. The Viral Darwinism of W32.Evol. VX Heaven. (Jan. 2006). Web. Retrieved 16 Oct. 2014, from http://vxheaven.org/lib/vor00.html

[25] Santanam, R. (2010). Cyber security, cyber crime and cyber forensics: Applications and perspectives: Applications and perspectives IGI Global

[26] Metamorphism and Self-Compilation in JavaScript. VX Heaven. (Dec. 2012). Web. Retrieved 04 Dec. 2014, from http://vxheaven.org/lib/vsp45.html#c1

[27] Greg Brail, Norris Boyd, Svein Atle, and Hannes Wallnöfer. Rhino Documentation. Mozilla Developer Network. Web. Retrieved 06 Oct. 2014, from https://developer.mozilla.org/en-US/docs/Rhino_documentation

[28] Transcriptase JavaScript Malware source. http://spth.virii.lu/Transcriptase.rar

[29] Prabhu, Amar. Working of Rhino JavaScript Engine. (2 Nov. 2012). Web. Retrieved 21 Oct. 2014, from http://www.quora.com/How-does-a-JavaScript-engine-work

[30] Abstract Syntax Tree. Wikipedia. Wikimedia Foundation. Web. Retrieved 09 Jan. 2015, from http://en.wikipedia.org/wiki/Abstract_syntax_tree

[31] Electrolysis/Firefox. MozillaWiki. Web. Retrieved 12 Feb. 2015, from https://wiki.mozilla.org/Electrolysis/Firefox

[32] Eric Shepherd, Kris Maglione, and Will Bamberg. XUL Tutorial. Mozilla Developer Network. Web. Retrieved 3 Sept. 2014, from https://developer.mozilla.org/en-US/Add-ons/Overlay_Extensions

[33] Will Bamberg, and Erik Vold. SDK and XUL Comparison. Mozilla Developer Network. Web. Retrieved 12 Sept. 2014, from https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/SDK_vs_XUL

[34] List of Most Popular Websites. Wikipedia. Wikimedia Foundation. Web. Retrieved 4 Apr. 2015, from http://en.wikipedia.org/wiki/List_of_most_popular_websites

APPENDIX

Code snippets

A.1 Python parser

Listing A.1: The parser code detects the valid start and end point of the JavaScript functions and properly splits the Transcriptase code.

```
with open("transcriptase.js", "r") as ins:
    total_functions = 1000
    line = ins.read()
    k = sys.argv[1]
    required = (int)(total_functions/k)
    cnt,braces,rbraces,sbraces,brackets_match,func_ind = 0,0,0,0,0,0
    skip, eachfun_done = False, False
    data, skip_char = '', ''
    function_start = True
    func = ['f','u','n','c','t','i','o','n']
    for c in line:
        if cnt == required and eachfun_done == True:
            cnt = 0
            eachfun_done = False
            #write data into a file
            data=''
        if (c=='\textquotedblleft' or c=="\textquoteleft") and skip==False:
            skip = True
            data = data+c
            skip_char = c
```

```
            continue
        if skip == True:
            data = data+c
            if skip_char == c:
                skip = False
                skip_char = ''
            continue
        if c == '(':
            rbraces+=1
        elif c == ')':
            rbraces-=1
        if c == '[':
            sbraces+=1
        elif c == ']':
            sbraces-=1
        if c == '{':
            if function_start==True:
                function_start=False
            braces+=1
        elif c == '}':
            braces-=1
        if braces == 0 and sbraces ==0 and rbraces == 0:
            if function_start==False:
                eachfun_done = True
        else:
            data = data +c
            continue
```

```
48          if func[func_ind] == c:
49              func_ind+=1
50          else:
51              func_ind=0
52          if func_ind == 8:
53              total_functions+=1
54              cnt+=1
55              function_start=True
56              eachfun_done = False
57              func_ind=0
58          data+=c
59      if data != "":
60          cnt = 0
61          eachfun_done = False
62          #write data into a file
63          data=''
```